Ogen: An Overlapping Grid Generator for Overture

William D. Henshaw [1]
CASC: Centre for Applied Scientific Computing[2]
Lawrence Livermore National Laboratory
Livermore, CA, 94551
henshaw@llnl.gov
http://www.llnl.gov/casc/people/henshaw
http://www.llnl.gov/casc/Overture

November 2, 2003

UCRL-MA-132237

Abstract:
We describe how to generate overlapping grids for use with Overture using the `ogen` program. The user must first generate `Mappings` to describe the geometry (a set of overlapping grids whose union covers the domain). The overlapping grid then is constructed using the `Ogen` grid generator. This latter step consists of determining how the different component grids interpolate from each other, and in removing grid points from holes in the domain, and removing unnecessary grid points in regions of excess overlap. This document includes a description of commands, presents a series of command files for generating various overlapping grids and describes the overlapping grid algorithm. The `ogen` program can also be used to build unstructured hybrid grids where the overlap is replaced by an unstructured grid.

Contents

[1]This work was partially supported by grant N00014-95-F-0067 from the Office of Naval Research
[2]Management prefers the spelling ‘Center’

1 Introduction

The ogen program can be used to interactively generate overlapping grids.

The basic steps to follow when creating an overlapping grid are

- create mappings that cover a domain and overlap where they meet.
- generate the overlapping grid (ogen calls the grid generator Ogen).
- save the grid in a data-base file.

The `ogen` program is found in the `Overture/bin` directory. Just type `ogen` to get started. You can also type `'ogen noplot'` in which case `ogen` will run without graphics. This is useful if you just want to execute a command file to regenerate a grid – running without graphics is faster. If you have a command file, `example.cmd`, then you can type `'ogen example.cmd'` or `'ogen example'` (a `.cmd` will automatically be added) to run the commands in the file. To run without graphics type `'ogen noplot example'`.

Once you have made a grid and saved it in a data-base file (named `myGrid.hdf`, for example) you can look at it using the command `Overture/bin/plotStuff myGrid.hdf` (or just `Overture/bin/plotStuff myGrid`).

Figure 1 shows a snap-shot of `ogen` running.

Other documents of interest that are available through the Overture home page are

- Mapping class documentation : `mapping.tex`, [2]. Many of the mappings that are used to create an overlapping grid are documented here.
- Interactive plotting : `PlotStuff.tex`, [3].

2 Commands

2.1 Commands for ogen

The commands in the initial `ogen` menu are

create mappings : create mappings to represent the geometry. See section (2.2).

generate an overlapping grid : once mappings have been created an overlapping grid can be generated with this option. This will call the Ogen grid generator. See section (9.1) for a list of the commands available with the grid generator.

make an overlapping grid : this calls the old Cgsh grid generator, the original Overture grid generator.

save and overlapping grid : Save an overlapping grid in a data base file.

2.2 Commands when creating Mappings

The basic commands available from the `create mappings` menu option are (this list will in general be out of date so you are advised to run `ogen` to see the currently available options). Most of these commands simply create a new Mapping and call the update function for that Mapping. Descriptions of the **Mapping's** referred to here can be found in the mapping documentation [2].

help : output minimal help.

1D Mappings :

- **line** : Build a line in 1D. This can be used for a 1D overlapping grid. Reference `LineMapping`.
- **stretching function** : Reference `StretchMapping`.
- **spline (1D)** : Reference `SplineMapping`.

2D Mappings :

- **Airfoil** : Build a two-dimensional airfoil from various choices including the NACA 4 digit series airfoils. Reference `AirfoilMapping`.
- **Annulus** : Reference `AnnulusMapping`.

Circle or ellipse : Reference `CircleMapping`.

DataPointMapping : Build a new Mapping from a set of discrete data points. The data points may be read from a plot3d file. Reference `DataPointMapping`.

line (2D) : Reference `LineMapping`.

nurbs (curve) : build a NURBS (a type of spline) curve or surface from control points or by interpolating data points. Reference `NurbsMapping`.

rectangle : Reference `SquareMapping`.

SmoothedPolygon : Build a grid or curve with a boundary that is a polygon with smoothed out corners. Reference `SmoothedPolygonMapping`.

spline : Reference `SplineMapping`.

tfi : Build a new Mapping from existing curves or surfaces using transfinite interpolation (Coon's patch). Reference `TFIMapping`.

3D Mappings :

Box : Reference `BoxMapping`.

Cylinder : Reference `CylinderMapping`.

Circle or ellipse (3D) : Reference `CircleMapping`.

CrossSection : Reference `CrossSectionMapping`.

DataPointMapping : Build a new Mapping from a set of discrete data points. The data points may be read from a plot3d file. Reference `DataPointMapping`.

line (3D) : Reference `LineMapping`.

nurbs (surface) :build a NURBS (a type of spline) curve or surface from control points or by interpolating data points. Reference `NurbsMapping`.

plane or rhombus : Reference `PlaneMapping`.

Sphere : Reference `SphereMapping`.

spline (3D) : Reference `SplineMapping`.

tfi : Build a new Mapping from existing curves or surfaces using transfinite interpolation (Coon's patch). Reference `TFIMapping`.

transform :

body of revolution : create a body of revolution from a two-dimensional Mapping. Reference `RevolutionMapping`.

elliptic : generate an elliptic grid on an existing grid in order to redistribute grid points. Reference `EllipticTransform`.

fillet : Build a fillet surface to join two intersecting surfaces. Reference `FilletMapping`.

hyperbolic : Reference `HyperbolicMapping`.

hyperbolic surface : Reference `HyperbolicSurfaceMapping`.

intersection : Determine the intersection curve between two intersecting surfaces. Reference `IntersectionMapping`.

mapping from normals : Generate a new Mapping by extending normals from a curve or a surface. Reference `NormalMapping`.

reparameterize : reparameterize an existing Mapping by

1. restricting the domain space to a sub-rectangle (this would be used to create an refinement patch on an adaptuve grid)
2. remove a polar singularity by creating a new patch with an orthographic transform.

Reference `ReparameterizationTransform`, `OrthographicTransform` and `RestrictionMapping`.

rotate/scale/shift : transform an existing Mapping. Reference `MatrixMapping`.

stretch coordinates : stretch (cluster) the grid points in the coordinate directions. Reference `StretchTransform` and `StretchMapping`.

change :

change a mapping : Make changes to an existing Mapping.

copy a mapping : Make a copy of an existing Mapping.

delete a mapping : delete an existing Mapping.

data base :

open a data-base : open an Overture data-base file (new or old).

get from the data-base : read Mapping's from the data-base.

put to the data-base : save a Mapping in the data-base.

close the data-base : close the data-base.

save plot3d file : write a plot3d file.

read from file :

read plot3d file : read a plot3d formatted file and extract the grids. Each grid becomes a `DataPointMapping`.

read iges file : *experimental* read an IGES (Initial Graphics Exchange Specification) file such as created by pro/ENGINEER and build NURBS and trimmed NURBS found in the file.

read overlapping grid file : read an existing overlapping grid data base file and extract all the Mapping's from it. These Mappings can then be changed.

view mappings : view the currently defined Mappings.

check mapping : check a Mapping to see that it is defined properly. This is normally only done when one defines a new Mapping.

exit this menu :

3 Things you should know to make an overlapping grid

Here are some things that you will need to know when building overlapping grids. The examples that follow will demonstrate all of these ideas.

3.1 Boundary conditions

Each side of each component grid must be given a boundary condition value. These boundary conditions are essential since they indicate whether a boundary is a physical boundary (a value greater than 0), an interpolation boundary (a value equal to zero) or a side that is has a periodic boundary condition (a value less than zero). The boundary condition values are stored in an array as

$$\mathtt{boundaryCondition}(\mathtt{side},\mathtt{axis}) = \begin{cases} >0 & \text{physical boundary} \\ =0 & \text{interpolation boundary} \\ <0 & \text{periodic boundary} \end{cases}$$

$$\begin{aligned}
\mathtt{boundaryCondition}(0,0) &= \text{left} \\
\mathtt{boundaryCondition}(1,0) &= \text{right} \\
\mathtt{boundaryCondition}(0,1) &= \text{bottom} \\
\mathtt{boundaryCondition}(1,1) &= \text{top} \\
\mathtt{boundaryCondition}(0,2) &= \text{front (3D)} \\
\mathtt{boundaryCondition}(1,2) &= \text{back (3D)}
\end{aligned}$$

where `side=0,1` and `axis=0,1` in 2D, or `axis=0,1,2` in 3D, indicates the face of the the grid. Note that each grid is a mapping from the unit square or unit cube to a physical domain – the terms left, right, bottom, top, front and back refer to the sides of the unit square or cube. When you enter the boundary condition values (when changing them in a mapping) you should enter them in the order: left, right, bottom, top, front, back.

The grid generator uses physical boundaries to cut holes in other grids that happen to cross that physical boundary. See, for example, the “cylinder in a channel example” where the rectangular grid has a hole cut out of it. Interpolation boundaries are non-physical boundaries where the grid generator will attempt to interpolate the points from other component grids. A periodic boundary can be either be a branch cut (as on an annulus) or it can indicate a periodic domain (as with a square where the right edge of the square is to be identified with the left edge).

3.2 Share flag

The share flag is used to indicate when two different component grids share a common boundary (see the “inlet outlet” example, section (4.7). The grid generator uses the share flag so that a boundary of one component grid will not accidently cut a hole in another grid when the two grids are actually part of the same boundary. This could happen since, due to inaccuracies in representing each grid, it may seem that the boundary on one grid lies inside or outside the other grid (even though they are meant to be the same boundary curve).

The share flag is saved in an array that is the same shape as the `boundary condition` array

$$
\begin{aligned}
\mathtt{share}(\mathtt{side},\mathtt{axis}) > 0 & \quad \text{a code that should be the same on all shared boundaries.} \\
\mathtt{share}(0,0) &= \text{left} \\
\mathtt{share}(1,0) &= \text{right} \\
\mathtt{share}(0,1) &= \text{bottom} \\
\mathtt{share}(1,1) &= \text{top} \\
\mathtt{share}(0,2) &= \text{front (3D)} \\
\mathtt{share}(1,2) &= \text{back (3D)}
\end{aligned}
$$

where `side=0,1` and `axis=0,1` in 2D, or `axis=0,1,2` in 3D, indicates the face of the the grid.

Thus the share flags on all grid faces that belong to the same boundary should be given the same share value. This could be accomplished by setting all `share` values to 1 say, although this is slightly dangerous as the grid generator could make a mistake. It is better to use a different positive integer for each different boundary.

3.3 Turning off the cutting of holes

By default, the overlapping grid generator will use any physical boundary (a side of a grid with a positive `boundaryCondition` to try and cut holes in any other grid that lies near the physical boundary. Thus in the “cylinder in a channel example” section (4.3) the inner boundary of the annulus cuts a hole in the rectangular grid. Sometimes, as in the “inlet outlet” example, section (4.7), one does not want this to happen. In this case it is necessary to explicitly specify which grids are allowed to cut holes in which other grids. This can be done through in the `change parameters` option with the `prevent hole cutting` option, see section the “inlet outlet” example, (4.7).

3.4 Turning off interpolation between grids

By default all grids can interpolate from all other grids. This default can be changed and you may specify which grids may interpolate from which other grids. This option can be used, for example, to build grids for two disjoint domains that match along a boundary as shown in figure (22).

3.5 Implicit versus explicit interpolation

There are two types of interpolation, **explicit** and **implicit**. **Explicit** interpolation means that a point that is interpolated will only use values on other grids that are not interpolation points themselves. This means that will the default 3 point interpolation the amount of overlap must be at least 1.5 grid cells wide. With explicit interpolation the interpolation equations can be solved explicitly (and this faster).

With **implicit** interpolation the points used in the interpolation stencil may themselves be interpolation points. This means that will the default 3 point interpolation the amount of overlap must be at least $.5$ grid cells wide. Thus **implicit interpolation is more likely to give a valid grid** since it requires less overlap. With implicit interpolation the interpolation equations are a coupled system that must be solved. This is a bit slower but the Overture interpolation function handles this automatically.

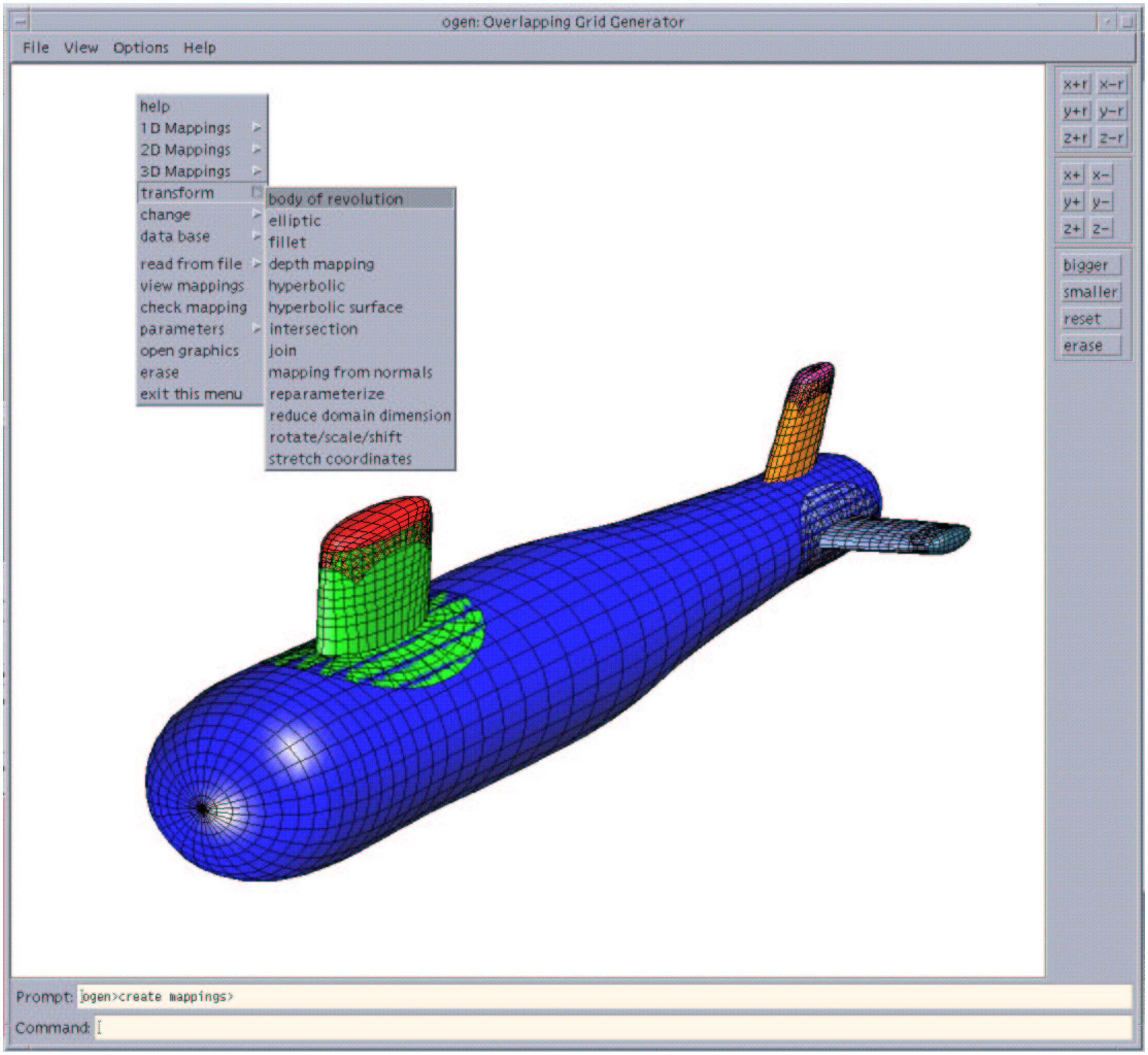


Figure 1: A snapshot of ogen

4 Examples

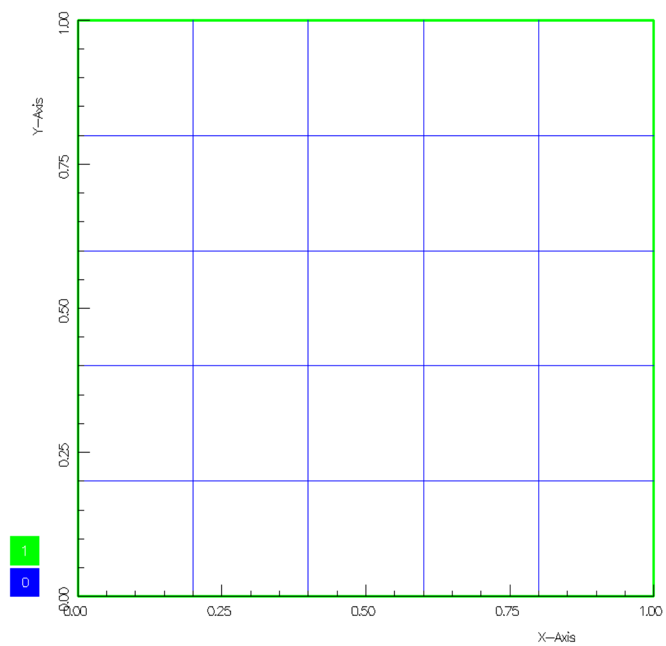
In this section we describe a number of *command files* that can be used to create various overlapping grids. During an interactive session a command file can be saved, see the option `'log commands to file'` in the `file` pull-down menu. By default the command file `ogen.cmd` is automatically saved. The command file will record all the commands that are issued. The command file can be later read in, using `'read command file'` in the `file` pull-down menu, and the commands will be executed. You can also type `'ogen example.cmd'` to run the command file named `example.cmd` with graphics or `'ogen noplot example.cmd'` to run without graphics.

The command file can be edited and changed. Once a complicated grid has been created it is usually easiest to make minor changes by editing the command file. The `pause` command can be added to the command file which will cause the program to pause at that point and wait for an interactive response – one can then can either `continue` or `break`.

4.1 Square

Here is a command file to create a square. (file `Overture/sampleGrids/square5.cmd`) We first make a mapping for the square and assign various parameters such as the number of grid points and the boundary conditions. Any positive number for the boundary condition indicates a physical boundary. Next the overlapping grid generator is called (`make an overlapping grid`) to make an overlapping grid (which is trivial in this case). Finally the overlapping grid is saved in a data-base file. The data-base file is an HDF formatted file. HDF is the the Hierarchical Data Format (HDF) from the National Centre for Super-Computing Applications (NCSA). You can look at the data base file created here by typing `plotStuff square5.hdf` (or just `plotStuff square5`) where `plotStuff` is found in `Overture/bin`.

```
* make a simple square
create mappings
  rectangle
    mappingName
      square
    lines
      6 6
    boundary conditions
      1 1 1 1
  exit
exit
*
generate an overlapping grid
  square
  done
  change parameters
    ghost points
      all
      2 2 2 2 2 2
  exit
  compute overlap
exit
*
save an overlapping grid
  square5.hdf
  square5
exit
```



An “overlapping grid” that is just a square

4.2 Stretched Annulus

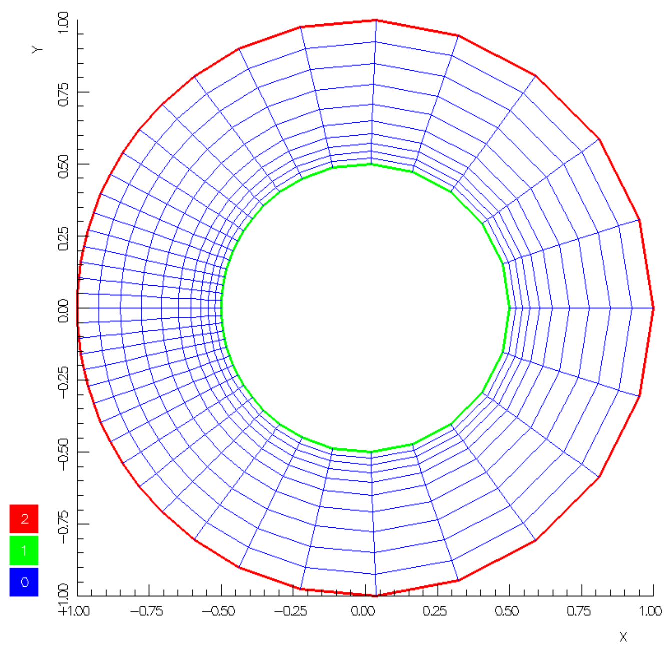
FAQ : What the heck is going on with the stretching function?! (F. Olsson-Hector)

Answer: Here is a command file to create an annulus with stretching. (file `Overture/sampleGrids/stretchedAnnulus.cmd`) Grid lines can be stretched in the coordinate directions (i.e. in the unit-square coordinates). When grid lines are stretched, as in the example below, the graphics screen will show one of the following

- The mapping to be stretched (annulus)
- The unit square to be stretched.
- The one dimensional stretching function.

The stretching functions are described in the documentation on Mapping's [2].

```
*
*  Create an annulus and stretch the grid lines
*
create mappings
* create an Annulus
  Annulus
  lines
    41 11
  exit
* stretch the grid lines
  stretch coordinates
    transform which mapping?
      Annulus
    stretch
      specify stretching along axis=0
* choose a layer stretching a*tanh(b*(r-c
        layers
          1
*         give a,b,c in above formula
          1. 10. .5
        exit
      specify stretching along axis=1
        layers
          1
          1. 5. 0.
      exit
    exit
  exit
exit this menu
*
* make an overlapping grid
*
generate an overlapping grid
  stretched-Annulus
  done
  compute overlap
  exit
*
* save as an hdf file
*
save an overlapping grid
stretchedAnnulus.hdf
grid
exit
```



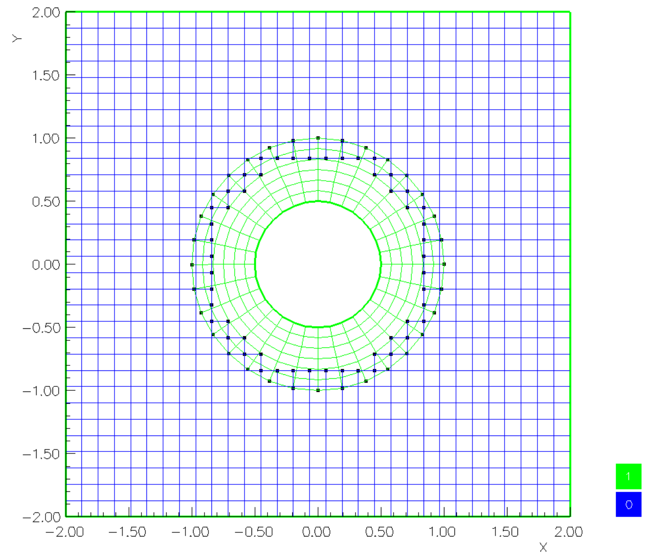
An annulus with stretching

For the pundits: The stretched annulus is a `StretchTransform` Mapping which is a composition of the `StretchedSquare` Mapping and the `Annulus` Mapping. The `StretchedSquare` uses the `Stretch` Mapping where the actual one dimensional stretching functions are defined.

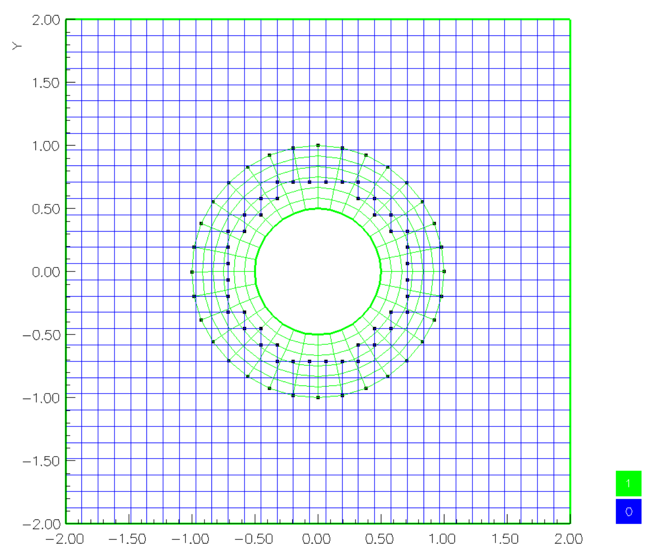
4.3 Cylinder in a channel

Here is a command file to create a cylinder in a channel. (file `Overture/sampleGrids/cic.cmd`) In this case we make two mappings, one a background grid and one an annulus. The boundary conditions on the annulus are set so that the outer boundary is an interpolation boundary (=0) while the boundary conditions on the branch cut are -1 to indicate a periodic boundary. We show two overlapping grids, one made with implicit interpolation (default) and one made with explicit interpolation. The latter has a bigger region of overlap.

```
*
* circle in a channel
*
create mappings
*
rectangle
  set corners
    -2. 2. -2. 2.
  lines
    32 32
  boundary conditions
    1 1 1 1
  mappingName
  square
exit
*
Annulus
  lines
    33 7
*  centre
*    0. 1.
  boundary conditions
    -1 -1 1 0
exit
*
exit
generate an overlapping grid
    square
    Annulus
  done
  change parameters
    * choose implicit or explicit interpolati
    * interpolation type
    *   implicit for all grids
    ghost points
      all
        2 2 2 2 2 2
  exit
*  display intermediate results
  compute overlap
  exit
*
save an overlapping grid
cic.hdf
cic
exit
```



An overlapping grid for a cylinder in a channel with implicit interpolation



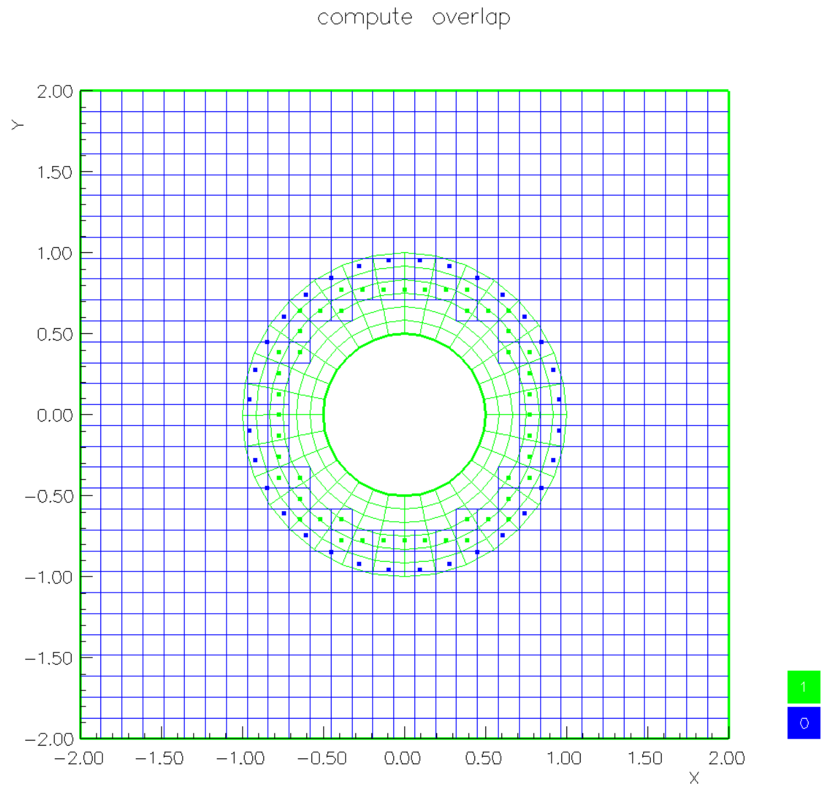
An overlapping grid for a cylinder in a channel with explicit interpolation

4.4 Cylinder in a channel, cell-centered version

Here we repeat the last example but create a cell-centered grid. In a cell-centered grid the cell-centres of one grid are interpolated from the cell-centres of another grid. For this reason the cell-centred grid requires slighly more overlap between the component grids.

```
*
* circle in a channel, cell centered grid
*
create mappings
*
rectangle
  set corners
    -2. 2. -2. 2.
  lines
    32 32
  boundary conditions
    1 1 1 1
  mappingName
  square
exit
*
Annulus
  lines
    33 7
  boundary conditions
    -1 -1 1 0
exit
*
exit
generate an overlapping grid
    square
    Annulus
  done
  change parameters
    * make the grid cell-centered
    cell centering
      cell centered for all grids
    exit
  compute overlap
  exit
*
save an overlapping grid
cicCC.hdf
cicCC
exit

```

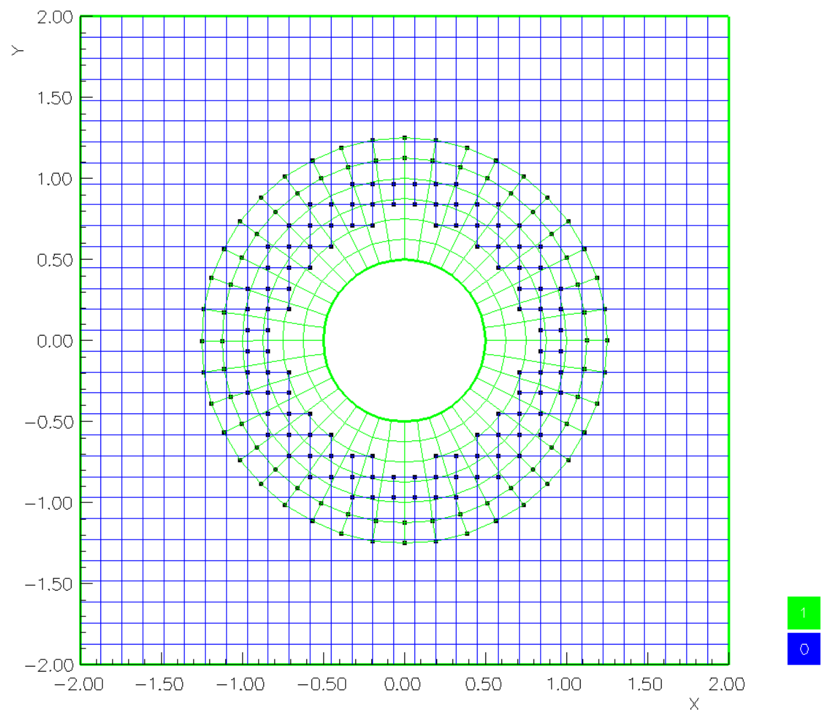


An overlapping grid for a cylinder in a channel, cell-centered case.

4.5 Cylinder in a channel, fourth-order version

Here we repeat the last example but create a grid appropriate for a fourth-order discretization. We need to increase the discretization width to 5 and the interpolation width to 5. This can either be done explicitly or the option "`order of accuracy`" can be used. Notice that two lines of interpolation points are generated as required by the wider stencil.

```
*
* circle in a channel, for fourth order accuracy. This
* can be used with primer/wave
*
create mappings
*
rectangle
  set corners
    -2. 2. -2. 2.
  lines
    129 129
  boundary conditions
    1 1 1 1
  mappingName
  square
exit
*
Annulus
  lines
    161 9
  outer radius
    .75
  boundary conditions
    -1 -1 1 0
exit
*
exit
generate an overlapping grid
    square
    Annulus
  done
  change parameters
    * choose implicit or explicit interp
    interpolation type
      implicit for all grids
      * explicit for all grids
    ghost points
      all
      2 2 2 2
    order of accuracy
      fourth order
*   we could also do the following:
*     discretization width
*      all
*      5 5
*     interpolation width
*      all
*      all
*      5 5
  exit
  compute overlap
  exit
*
save an overlapping grid
cic.4.hdf
cic4
exit
```

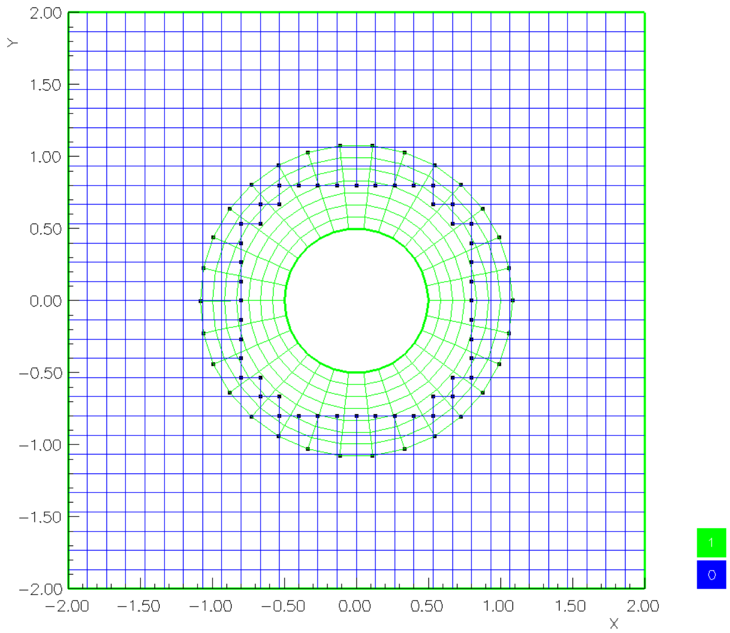


An overlapping grid for a cylinder in a channel, fourth-order case.

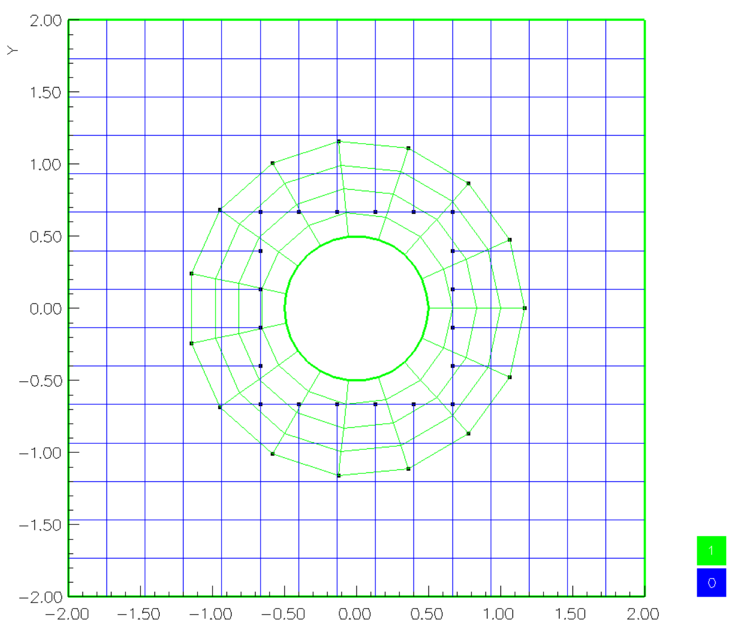
4.6 Cylinder in a channel, multigrid version

Here we make a grid that can be used with a multigrid solver. The only difference in the command file is that we must specify how many multigrid levels we require. **NOTE** that since each multigrid level must be a valid overlapping grid you cannot expect to have more than a few levels. See the examples in the primer for how to access the different multigrid levels in a `CompositeGrid`.

```
*
* circle in a channel with MG levels
*
create mappings
*
rectangle
  set corners
    -2. 2.  -2. 2.
  lines
    45 45
  boundary conditions
    1 1 1 1
  mappingName
  square
exit
*
Annulus
  lines
    65 9
  boundary conditions
    -1 -1 1 0
exit
*
exit
generate an overlapping grid
  specify number of multigrid levels
    2
  square
  Annulus
  done
  change parameters
    interpolation type
      explicit for all grids
    ghost points
      all
      2 2 2 2 2 2
  exit
  * pause
  compute overlap
exit
save an overlapping grid
cicmg.hdf
cic
exit
```



An overlapping grid for a cylinder in a channel, multigrid level 0.



An overlapping grid for a cylinder in a channel, multigrid level 1.

4.7 Inlet-outlet

In this example we demonstrate

share flags: to specify that two component grids have sides that belong to the same physical boundary curve. This prevents one physical boundary from accidently cutting a hole on a grid that shares the same boundary.

no hole cutting: turn off hole cutting to prevent physical boundaries from cutting holes in some other grids.

view mappings: the mappings can be plotted with boundaries coloured by the boundary condition values or coloured by the share flag values. This allows one to check that the values have been set properly.

This grid is remarkably similar to a grid created by Anders Petersson.

Here is a command file to create the grid for the inlet-outlet example. (file `Overture/sampleGrids/inletOutlet.cmd`).

```
*
* create a grid to demonstrate various features
*
create mappings
  * make a back ground grid
  rectangle
    set corners
      0 2.  0 1.
    lines
      61 31
    mappingName
     backGroundGrid
    share
      1 2 3 4
  exit
  * make an annulus
  Annulus
    centre for annulus
      1. .5
    inner radius
     .2
    outer radius
     .4
    lines
      41 9
    mappingName
      annulus
    boundary conditions
      -1 -1 1 0
  exit
  * the inlet (on the right) will consist of two
  * smoothed polygons
  SmoothedPolygon
    mappingName
      inlet-top
    vertices
     3
     2. .85
     2. .65
     2.25 .65
   n-dist
     fixed normal distance
     -.175  .2
   sharpness
     10.
     10.
     10.
   t-stretch
     0. 10.
     1. 10.
     0. 10.
   lines
     25 11
   boundary conditions
     0 1 1 0
   * One boundary here should match one boundary of
   * the backGroundGrid, while another boundary
   * should match a boundary on the inlet-bottom.
   * Set share flag to match corresponding share values
   share
     0 5 2 0
   exit
*
  SmoothedPolygon
    mappingName
      inlet-bottom
    vertices
     3
     2. .15
     2. .35
     2.25 .35
   lines
     25 11
   n-dist
     fixed normal distance
      .175  .2
   sharpness
     10.
     10.
     10.
   t-stretch
     0. 10.
     1. 10.
     0. 10.
   boundary conditions
     0 1 1 0
   * One boundary here should match one boundary
   * of the backGroundGrid, while another boundary
   * should match a bounbdary on the inlet-bottom.
   * Set share flag to match corresponding share values
   share
     0 5 2 0
   exit
  * here is an outlet grid made in the poor man’s way
  rectangle
    set corners
      -.35 .05  .3 .7
    lines
      15 15
    mappingName
     outlet
    boundary conditions
      1 0 1 1
  exit
  * now look at the mappings
  view mappings
    backGroundGrid
    annulus
```

```
    inlet-top
    inlet-bottom
    outlet
    *
    * The grid is plotted with boundaries coloured
    *  by the boundary condition number. Here we
    * should check that all interpolation boundaries
    * are 0 (blue), all physical boundaries are positive
    * and periodic boundaries are black
    * pause
    *
    * now we plot the boundaries by share value
    * The sides that correspond to the same boundary
    * should be the same colour
    colour boundaries by share value
    * pause
    erase and exit
  exit
generate an overlapping grid
* put the nonconforming grid first to be a lower
* priority than the back-ground
    outlet
    backGroundGrid
    annulus
    inlet-top
    inlet-bottom
  done
  change parameters
    prevent hole cutting
      backGroundGrid
        all
      outlet
        all
    done
    ghost points
      all
      2 2 2 2 2 2
  exit
*  display intermediate
*  set debug parameter
*    31
  compute overlap
  exit
*
save an overlapping grid
  inletOutlet.hdf
  inletOutlet
exit
```

The cell-centred version may be created with `Overture/sampleGrids/inletOutlet.cmd`.

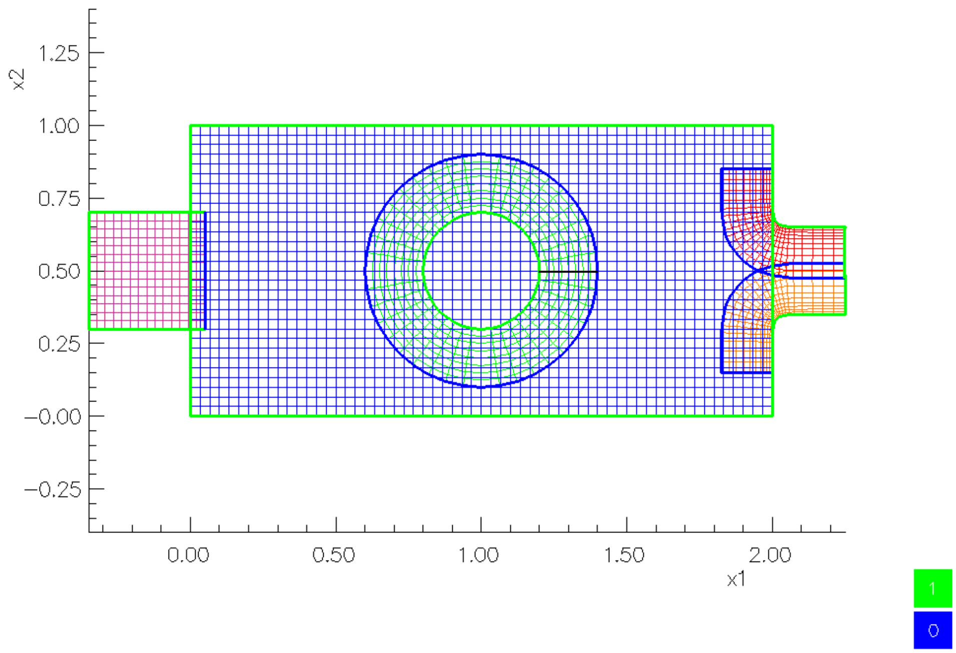


Figure 2: Inlet-outlet mappings plotted from the "view mappings" menu, showing boundary condition values. Physical boundaries have a positive value (1=green), interpolation boundaries have a value of zero (0=blue) and periodic boundaries have a negative value (shown in black).

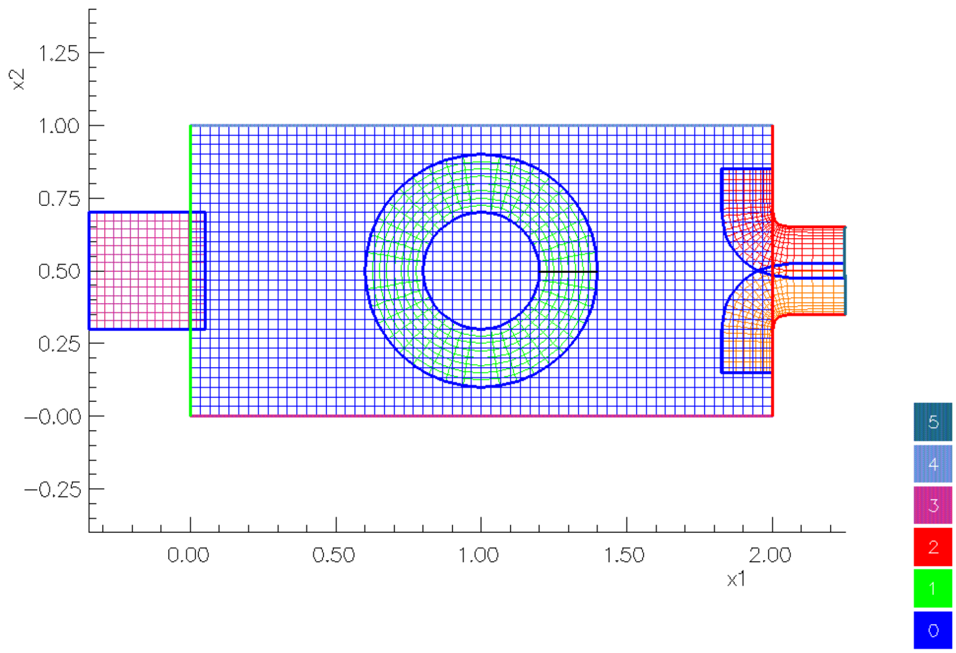


Figure 3: Inlet-outlet mappings plotted from the “view mappings” menu, showing shared side values. Grids that share the same physical boundary should have the same value of the share flag. For example, the two inlet grids on the right share boundaries with the back-ground grid (value 2=red). The inlet grids also share boundaries with each other (value 5)

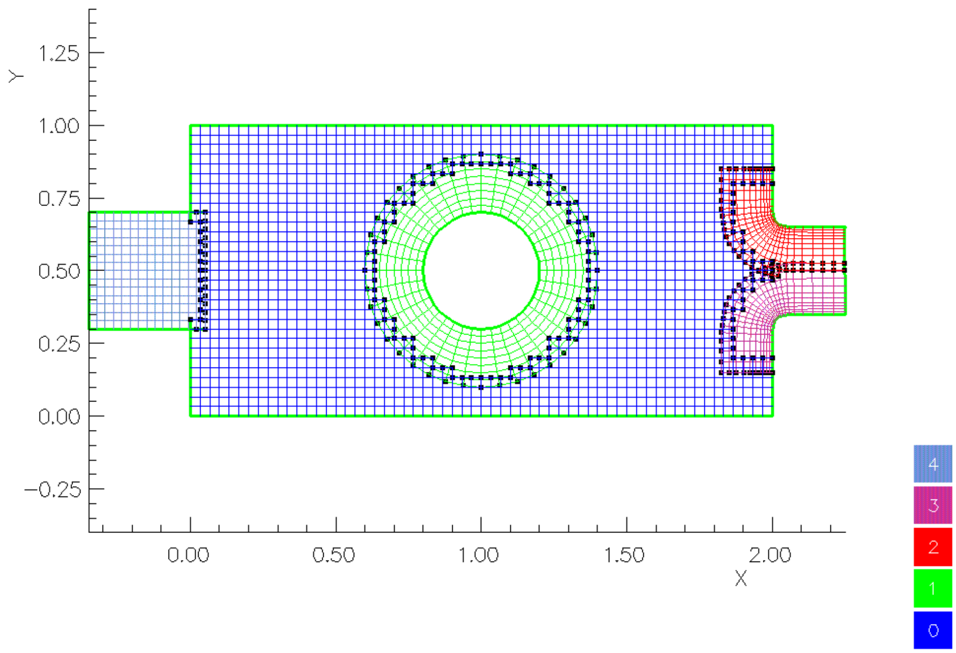


Figure 4: Inlet-outlet overlapping grid. To create this grid we had to prevent the background grid from cutting holes in the two inlet grids (on the right) and the outlet grid on the left. The outlet grid was also prevented from cutting holes in the background grid.

4.8 Valve

Here is a command file to create a grid around a two-dimensional valve (file `Overture/sampleGrids/valve.cmd`).

```
*
* Create an overlapping grid for a 2D valve
*
*  time to make: old:27s (ultra) new: 4.4s
*
create mappings
  *
  * First make a back-ground grid
  *
  rectangle
    mappingName
      backGround
    set corners
      0 1.  0 1.
    lines
      * 41 41
      * 51 51
      49 49
    share
      1 2 3 4
  exit
  *
  * Now make the valve
  *
  SmoothedPolygon
    mappingName
      valve
    vertices
    * .4 .4 .65 .65  ok
    * .45 .45 .7 .7  ok
    * .47  .47  .72  .72  ok
    * .475 .475 .725 .725 no
    * .47  .47  .72  .72  last used, ok
     4
     0.47  0.
     0.47  .75
     0.72  .5
     0.72  0.
    n-dist
      fixed normal distance
      * .1
      .05
    lines
      * 65 9
      * 75 9
      73 9
    boundary conditions
      1 1 1 0
    share
      3 3 0 0
    sharpness
      15
      15
      15
      15
    t-stretch
      1. 0.
      1. 6.
      1. 4.
      1. 0.
    n-stretch
      1. 4. 0.
  exit
  *
  * Here is the part of the boundary that
  * the valve closes against
  *
  SmoothedPolygon
    mappingName
      stopper
    vertices
      4
      1. .5
      0.75 .5
      0.5 .75
      0.5 1.
      n-dist
        fixed normal distance
        * .1
        .05
      lines
        * 61 9
        * 61 9
        65 9
      t-stretch
        1. 0.
        1. 5.
        1. 5.
        1. 0.
      n-stretch
        1. 4. 0.
      boundary conditions
        1 1 1 0
      share
        2 4 0 0
  exit
exit
*
* Make the overlapping grid
*
generate an overlapping grid
    backGround
    stopper
    valve
  done
  change parameters
    ghost points
      all
      2 2 2 2 2 2
  exit
*  debug
*    7
*  display intermediate results
  compute overlap
*  pause
  exit
*
* save an overlapping grid
save a grid (compressed)
valve.hdf
valve
exit
```

The resulting grid is shown in figure 5. The cell centered version may be created with `Overture/sampleGrids/valveCC.cmd`.

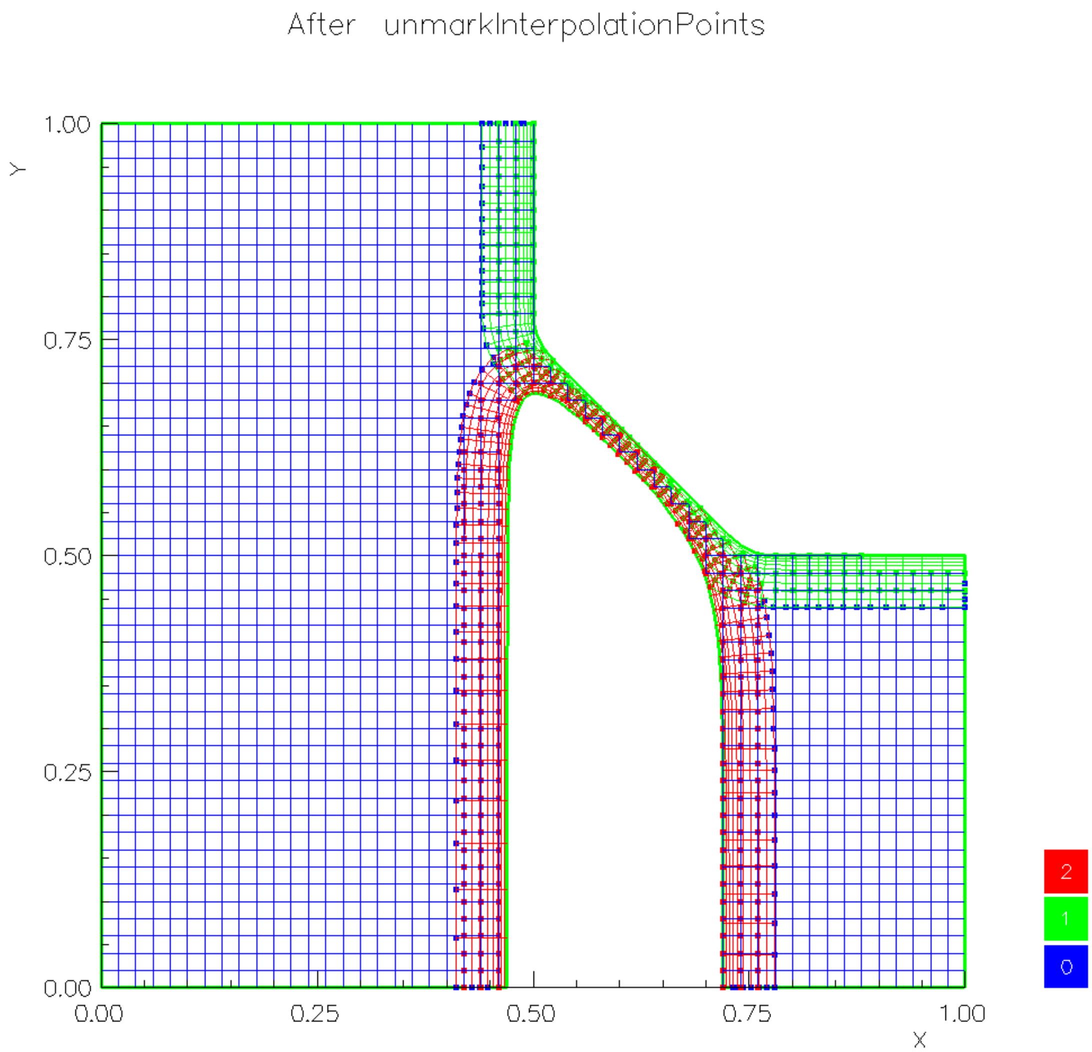


Figure 5: An overlapping grid for a valve

4.9 NACA airfoil

Here is a command file to create a grid around a two-dimensional NACA0012 airfoil (file `Overture/sampleGrids/naca0012.cmd`). The airfoil curve is created first with the `AirfoilMapping` (see the Mapping documentation for an explanation of NACA 4 digit airfoils). This curve is blended with an ellipse (using transfinite interpolation) to make an initial grid. The transfinite interpolation mapping then then smoothed using elliptic grid generation to form the airfoil grid.

```
*
* Make a grid around a NACA0012 airfoil
*
create mappings
  *
  * First make a back-ground grid
  *
  rectangle
    mappingName
      backGround
    set corners
      -1.5 2.5  -1.5 1.5
    lines
      41 33  41 31
  exit
  * make the NACA0012 airfoil (curve)
  Airfoil
    airfoil type
      naca
  exit
  * make an ellipse as an outer boundary
  Circle or ellipse
    specify centre
     .5 .0
    specify axes of the ellipse
      1.5 1.
  exit
  * blend the airfoil to the ellipse to make a grid
  tfi
    choose bottom curve
      airfoil
    choose top curve
      circle
    boundary conditions
      -1 -1 1 0
    lines
      73 17
    mappingName
      airfoil-tfi
    * pause
  exit
  *
  elliptic
    *project onto original mapping (toggle)
    transform which mapping?
      airfoil-tfi
  elliptic smoothing
    * slow start to avoid porblems at trailing edge
    number of multigrid levels
      3
    maximum number of iterations
      15
    red black
    smoother relaxation coefficient
      .1
    generate grid
    * now reset parameters for better convergence
    maximum number of iterations
      30
    smoother relaxation coefficient
      .8
    generate grid
    exit
    mappingName
      airfoil-grid
    * pause
  exit
exit
*
* make an overlapping grid
*
generate an overlapping grid
    backGround
    airfoil-grid
  done
  change parameters
    ghost points
      all
      2 2 2 2 2 2
  exit
  compute overlap
exit
*
save an overlapping grid
naca0012.hdf
naca
exit
```

The resulting grid is shown in figure 6.

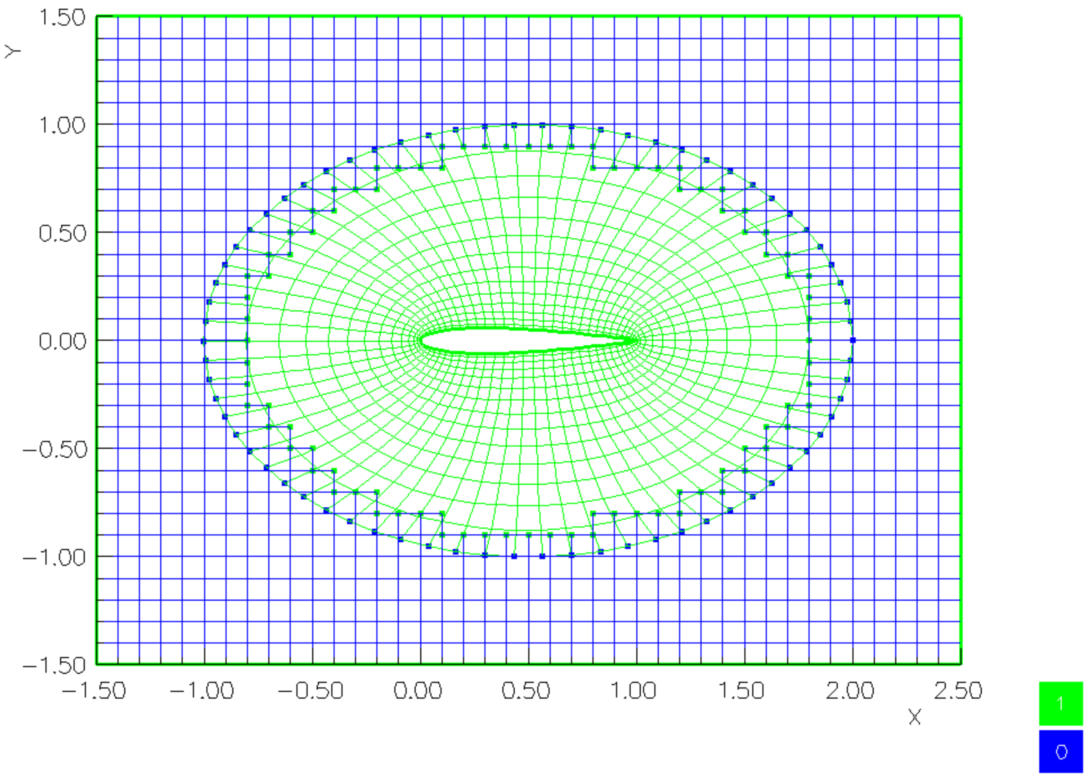


Figure 6: An overlapping grid for a NACA0012 airfoil

4.10 Hybrid grid for the inlet-outlet

Here is a command file to create a hybrid for an inlet-outlet geometry. `Overture/sampleGrids-/inletOutlet.hyb.cmd`).

```
*
* create a grid to demonstrate various features
*
create mappings
  * make a back ground grid
  rectangle
    set corners
      0 2.  0 1.
    lines
      61 31
    mappingName
     backGroundGrid
    share
      1 2 3 4
  exit
  * make an annulus
  Annulus
    centre for annulus
      1. .5
    inner radius
     .2
    outer radius
     .4
    lines
      41 9
    mappingName
      annulus
    boundary conditions
      -1 -1 1 0
  exit
  * the inlet (on the right) will consist of two
  * smoothed polygons
  SmoothedPolygon
    mappingName
      inlet-top
    vertices
     3
     2. .85
     2. .65
     2.25 .65
   n-dist
     fixed normal distance
     -.175  .2
   sharpness
     10.
     10.
     10.
   t-stretch
     0. 10.
     1. 10.
     0. 10.
   lines
     25 11
   boundary conditions
     0 1 1 0
   * One boundary here should match one boundary of
   * the backGroundGrid, while another boundary
   * should match a boundary on the inlet-bottom.
   * Set share flag to match corresponding share values
   share
     0 5 2 0
   exit
*
  SmoothedPolygon
    mappingName
      inlet-bottom
    vertices
     3
     2. .15
     2. .35
     2.25 .35
   lines
     25 11
   n-dist
     fixed normal distance
      .175  .2
   sharpness
     10.
     10.
     10.
   t-stretch
     0. 10.
     1. 10.
     0. 10.
   boundary conditions
     0 1 1 0
   * One boundary here should match one boundary
   * of the backGroundGrid, while another boundary
   * should match a bounbdary on the inlet-bottom.
   * Set share flag to match corresponding share values
   share
     0 5 2 0
   exit
  * here is an outlet grid made in the poor man's way
  rectangle
    set corners
      -.35 .05  .3 .7
    lines
      15 15
    mappingName
     outlet
    boundary conditions
      1 0 1 1
  exit
  * now look at the mappings
  view mappings
    backGroundGrid
    annulus
    inlet-top
    inlet-bottom
    outlet
    *
    * The grid is plotted with boundaries coloured
    *  by the boundary condition number. Here we
    * should check that all interpolation boundaries
    * are 0 (blue), all physical boundaries are positive
    * and periodic boundaries are black
    * pause
    *
    * now we plot the boundaries by share value
    * The sides that correspond to the same boundary
    * should be the same colour
    colour boundaries by share value
    pause
    erase and exit
  exit
generate a hybrid mesh
* put the nonconforming grid first to be a lower
* priority than the back-ground
    outlet
    backGroundGrid
    annulus
```

```
    inlet-top
    inlet-bottom
  done
  change parameters
    prevent hole cutting
      backGroundGrid
        all
      outlet
        all
    done
  exit
*  display intermediate
*  set debug parameter
*    31
  compute overlap
  exit
  set plotting frequency (<1 for never)
  -1
  continue generation
  exit
  save grid in ingrid format
  inletOutlet.hyb.msh
exit
*

```

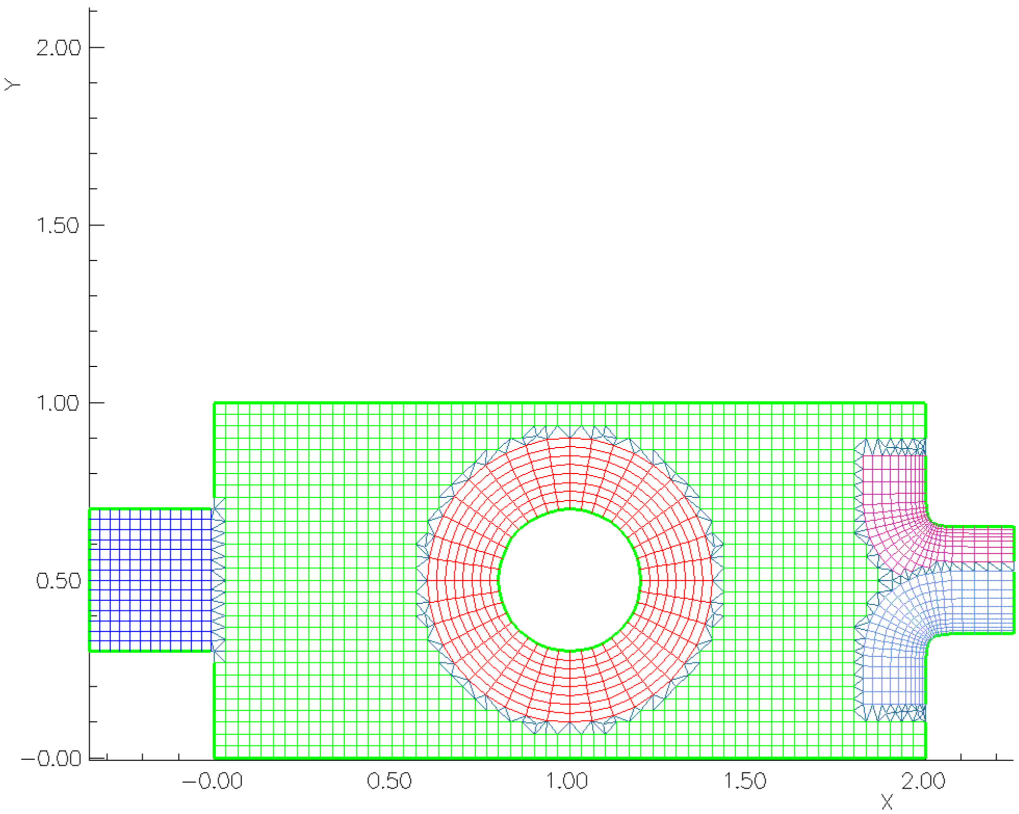
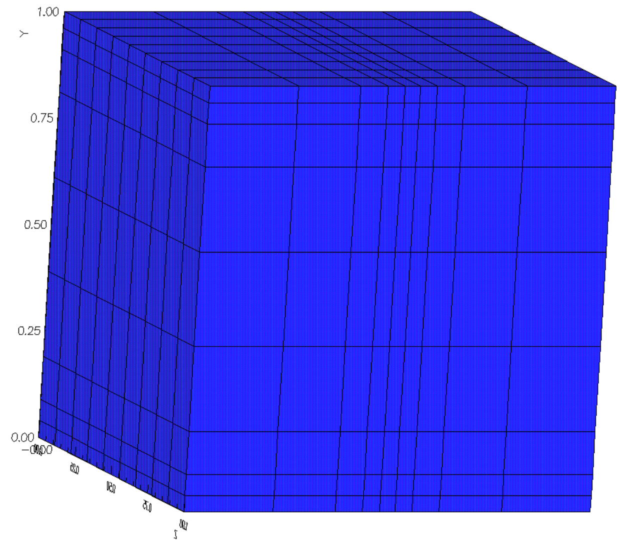


Figure 7: A hybrid grid for an inlet-outlet geometry.

4.11 Stretched cube

Here is a command file to create a simple box in 3D with stretched grid lines. (file `Overture/sampleGrids/-stretchedCube.cmd`)

```
*
* Create a 3D cube with stretched grid lines
*
create mappings
  Box
    exit
  stretch coordinates
    stretch
*     choose a layer stretching a*tanh(b*(r-
*     along axis 0
      specify stretching along axis=0 (x1)
        layers
        1
*         give a,b,c in above formula
        1. 10. .5
      exit
*     choose a stretching function with 2
*     layers along axis1
      specify stretching along axis=1 (x2)
        layers
        2
*         give a,b,c for layer 1
        1. 10. 0.
*         give a,b,c for layer 2
        1. 10. 1.
      exit
    exit
  exit
exit this menu
generate an overlapping grid
  stretched-box
  done
  compute overlap
exit
save an overlapping grid
  stretchedCube.hdf
  stretchedCube
exit
```



An overlapping grid for a stretched cube.

4.12 Sphere in a box

Here is a command file to create a sphere in a box. The sphere is covered with two orthographic patches, one for the north-pole and one for the south-pole. (file `Overture/sampleGrids/sib.cmd`)

```
*
* command file to create a sphere in a box
*
*  time to make: 594s new: 3.5
*     cpu=2s (ov15 sun-ultra optimized)
*        =.37 (tux50)
create mappings
* first make a sphere
Sphere
exit
*
* now make a mapping for the north pole
*
reparameterize
  orthographic
    specify sa,sb
      2.5 2.5
  exit
  lines
    15 15 5
  boundary conditions
    0 0 0 0 1 0
  share
    0 0 0 0 1 0
  mappingName
    north-pole
exit
*
* now make a mapping for the south pole
*
reparameterize
  orthographic
    choose north or south pole
      -1
    specify sa,sb
      2.5 2.5
  exit
  lines
    15 15 5
  boundary conditions
    0 0 0 0 1 0
  share
    0 0 0 0 1 0
  mappingName
    south-pole
exit
*
* Here is the box
*
Box
  set corners
    -2 2 -2 2 -2 2
  lines
    21 21 21
  mappingName
    box
  exit
exit
*
generate an overlapping grid
  box
  north-pole
  south-pole
  done
  change parameters
*    interpolation type
*      explicit for all grids
    ghost points
      all
      2 2 2 2 2 2
  exit
  compute overlap
exit
save an overlapping grid
sib.hdf
sib
exit
```

The resulting grid is shown in figure 8. The cell-centered version can be made with `Overture/sampleGrids/sibCC.cmd`.

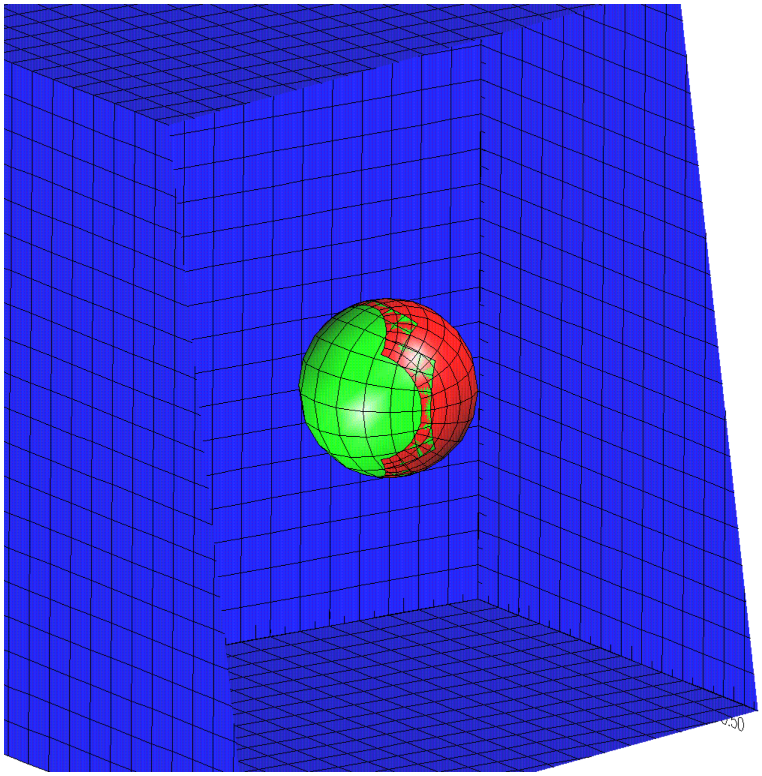


Figure 8: An overlapping grid for a sphere in a box. The sphere is covered with two patches.

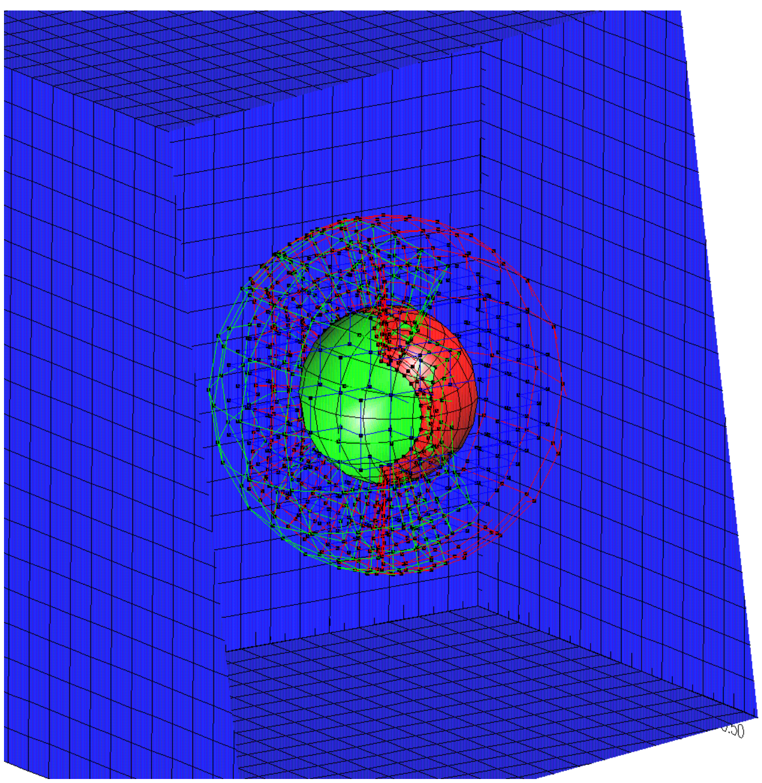


Figure 9: An overlapping grid for a sphere in a box. The interpolation points are also shown.

4.13 Sphere in a tube

Here is a command file to create a sphere in a cylindrical tube. The sphere is covered with two orthographic patches, one for the north-pole and one for the south-pole. The sphere is contained in a tube that is represented as a cylinderical annulus together with a rectangular box that forms the core of the cylinder. (file `Overture/sampleGrids/sphereInATube.cmd`)

```
*
* command file to create a sphere in cylindrical tube
*
*
create mappings
* first make a sphere
Sphere
exit
*
* now make a mapping for the north pole
*
reparameterize
  orthographic
    specify sa,sb
      2.5 2.5
  exit
  lines
    15 15 5
  boundary conditions
    0 0 0 0 1 0
  share
   0 0 0 0 1 0
  mappingName
    north-pole
exit
*
* now make a mapping for the south pole
*
reparameterize
  orthographic
    choose north or south pole
      -1
    specify sa,sb
      2.5 2.5
  exit
  lines
    15 15 5
  boundary conditions
    0 0 0 0 1 0
  share
    0 0 0 0 1 0
  mappingName
    south-pole
exit
*
* Here is the cylinder
*
  * main cylinder
  Cylinder
    mappingName
      cylinder
    * orient the cylinder so y-axis is axial direction
    orientation
      2 0 1
    bounds on the radial variable
      .3 .8
    bounds on the axial variable
      -1. 1.
    lines
      55 21 9
    boundary conditions
      -1 -1 2 3 0 4
    share
      0 0 2 3 0 0
  exit
* core of the main cylinder
  Box
    mappingName
      cylinderCore
    specify corners
    -.5 -1. -.5  .5 1. .5
    lines
      19 21 19
    boundary conditions
      0 0 2 3 0 0
    share
      0 0 2 3 0 0
  exit
*  pause
*
exit
generate an overlapping grid
  cylinderCore
  cylinder
  north-pole
  south-pole
  done
  change parameters
    ghost points
      all
      2 2 2 2 2 2
  exit
   * display intermediate
  compute overlap
  * continue
  * pause
exit
save an overlapping grid
sphereInATube.hdf
sit
exit
```

The resulting grid is shown in figure 10.

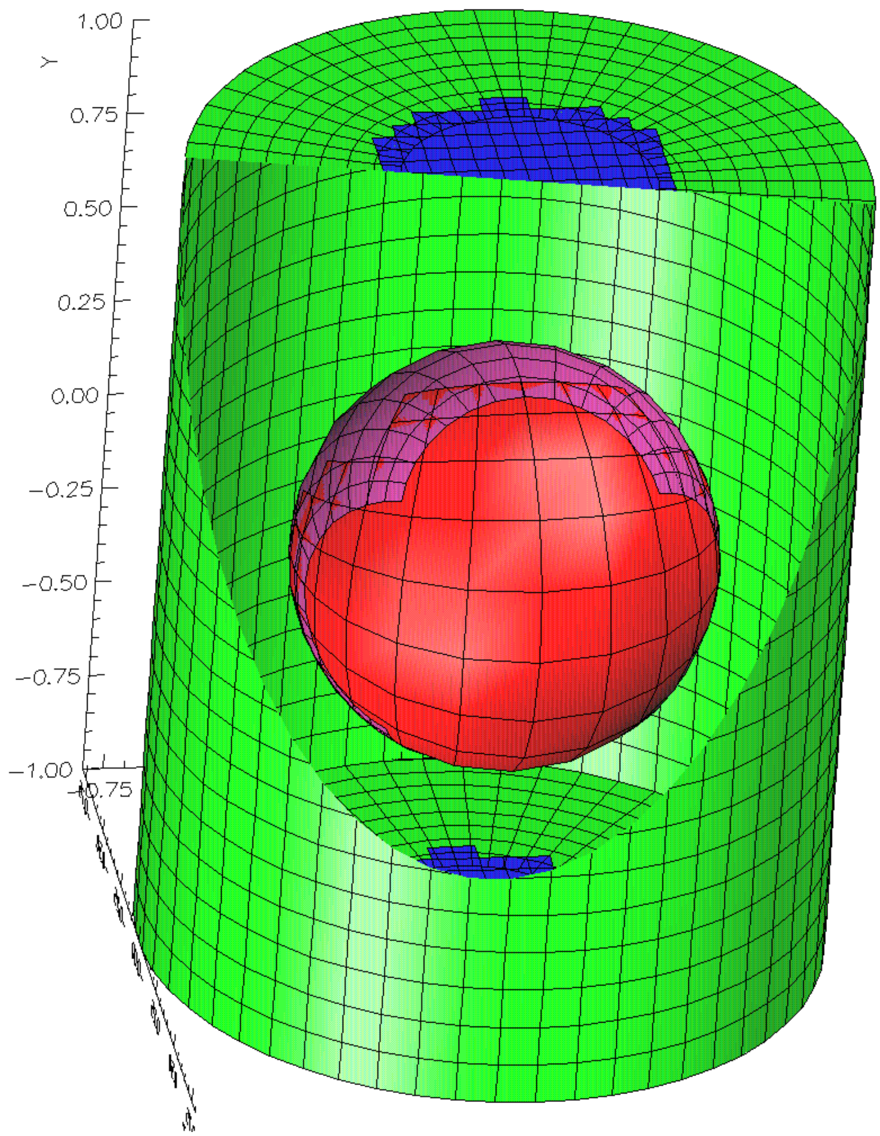


Figure 10: An overlapping grid for a sphere in a cylindrical tube

4.14 Intersecting pipes

Here is a command file to create a grid for two intersecting pipes. Each pipe is made from a cylindrical annulus with a rectangular grid for the core. The pipes intersect using the poor man's intersection method with non-conforming grids. (A more refined intersection would use a fillet). The key point here is that the boundaries must not cut holes and so this feature is turned off. (file `Overture/sampleGrids/pipes.cmd`)

```
*
* Make an overlapping grid for two intersecting pipes
*     cpu=2s (ov15 sun-ultra optimized)
*
create mappings
* Here is the main pipe
  Cylinder
    orientation
      1 2 0
    bounds on the radial variable
      .25 .5
    bounds on the axial variable
      -1.5 1.
    mappingName
      mainPipe
    lines
      25 21 7
    boundary conditions
      -1 -1 1 1 0 2
    share
      0 0 1 2 0 0
    exit
* Here is the core of the main pipe
*  note: there is trouble if corner of core just
*  sticks outside the main pipe -- hole cutter
*  misses. (happens with core half width= .3)
  Box
    specify corners
      -1.5 -.25 -.25 1. .25 .25
    lines
      21 9 9
    boundary conditions
      1 1 0 0 0 0
    mappingName
      mainCore
    share
      1 2 0 0 0 0
    exit
* Here is the branch pipe
  Cylinder
    orientation
      2 0 1
    bounds on the radial variable
      .2 .4
    bounds on the axial variable
      .25 1.25
    lines
      23 11 7    21  11 7
    boundary conditions
      -1 -1 0 1 0 2
    share
      0   0 0 3 0 0
    mappingName
      branchPipe
    exit
* Here is the core of the branch pipe
  Box
    specify corners
      -.25 .25 -.25 .25 1.25 .25
    lines
      9 15 9
    boundary conditions
      0 0 0 1 0 0
    share
      0 0 0 3 0 0
    mappingName
      branchCore
    exit
  exit
generate an overlapping grid
  branchCore
  branchPipe
  mainCore
  mainPipe
  done
  change parameters
    prevent hole cutting
      all
      all
    done
    allow hole cutting
      branchPipe
        branchCore
      mainCore
        mainPipe
    done
    ghost points
      all
      2 2 2 2 2 2
  exit
  * pause
  compute overlap
exit
save an overlapping grid
pipes.hdf
pipes
exit
```

The resulting grid is shown in figure 11.

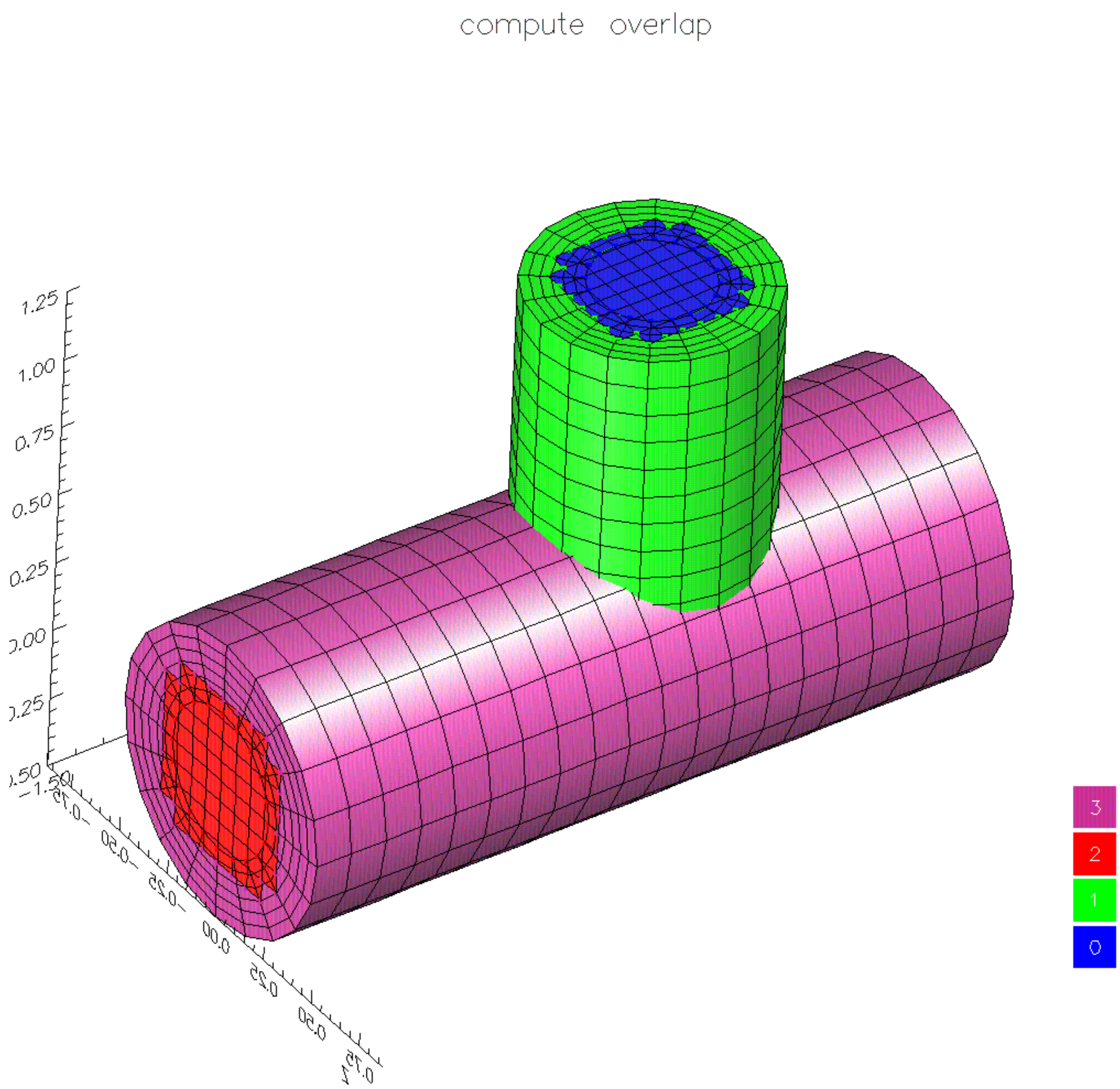


Figure 11: An overlapping grid for two intersecting pipes

4.15 Body Of Revolution

Here is a command file to create a grid for a body of revolution. The body of revolution is created by revolving a two-dimensional grid about a given line. The two dimensional grid in this case is created with the `SmoothedPolygon` Mapping. The body of revolution has a spherical polar singularity at both ends. We generate a new Mapping to cover each singularity. We reparameterize the ends using an orthographic transformation. (file `Overture/sampleGrids/revolve.cmd`)

```
*
* Create a cylindrical body of revolution
* from a Smoothed Polygon
*     cpu=48s (ov15 sun-ultra optimized)
create mappings
  SmoothedPolygon
    vertices
    7
    -1. 0.
    -1. .25
    -.8 .5
    0. .5
    .8 .5
    1. .25
    1. 0.
    n-dist
    fixed normal distance
    .1
    n-dist
    fixed normal distance
    .4
    corners
    specify positions of corners
    -1. 0.
    1. 0
    -1.4 0.
    1.4 0
    t-stretch
    0 5
    .15 10
    .15 10
    0 10
    .15 10
    .15 10
    0 10
  exit
* making a body of revolution
*  pause
  body of revolution
    tangent of line to revolve about
    1. 0 0
    mappingName
      cylinder
    lines
      55 25 7
    boundary conditions
      0 0 -1 -1 1 0
    share
      0 0  0  0 1 0
  exit
* patch on the front singularity
  reparameterize
    mappingName
      front
    lines
      15 15 5
    orthographic
      specify sa,sb
        .5 .5
    exit
    boundary conditions
      0 0 0 0 1 0
    share
      0 0 0 0 1 0
  exit
*   patch on back singularity
  reparameterize
    mappingName
      back
    lines
      15 15 7
    orthographic
      choose north or south pole
        -1
      specify sa,sb
        .5 .5
    exit
    boundary conditions
      0 0 0 0 1 0
    share
      0 0 0 0 1 0
  exit
*
* Here is the box
*
  Box
    specify corners
      -2 -1 -1 2 1 1
    lines
      61 31 31
    mappingName
      box
    exit
  * pause
exit
generate an overlapping grid
  box
  cylinder
  front
  back
  done
  compute overlap
exit
*
save an overlapping grid
revolve.hdf
revolve
exit
```

The resulting grid is shown in figure 12.

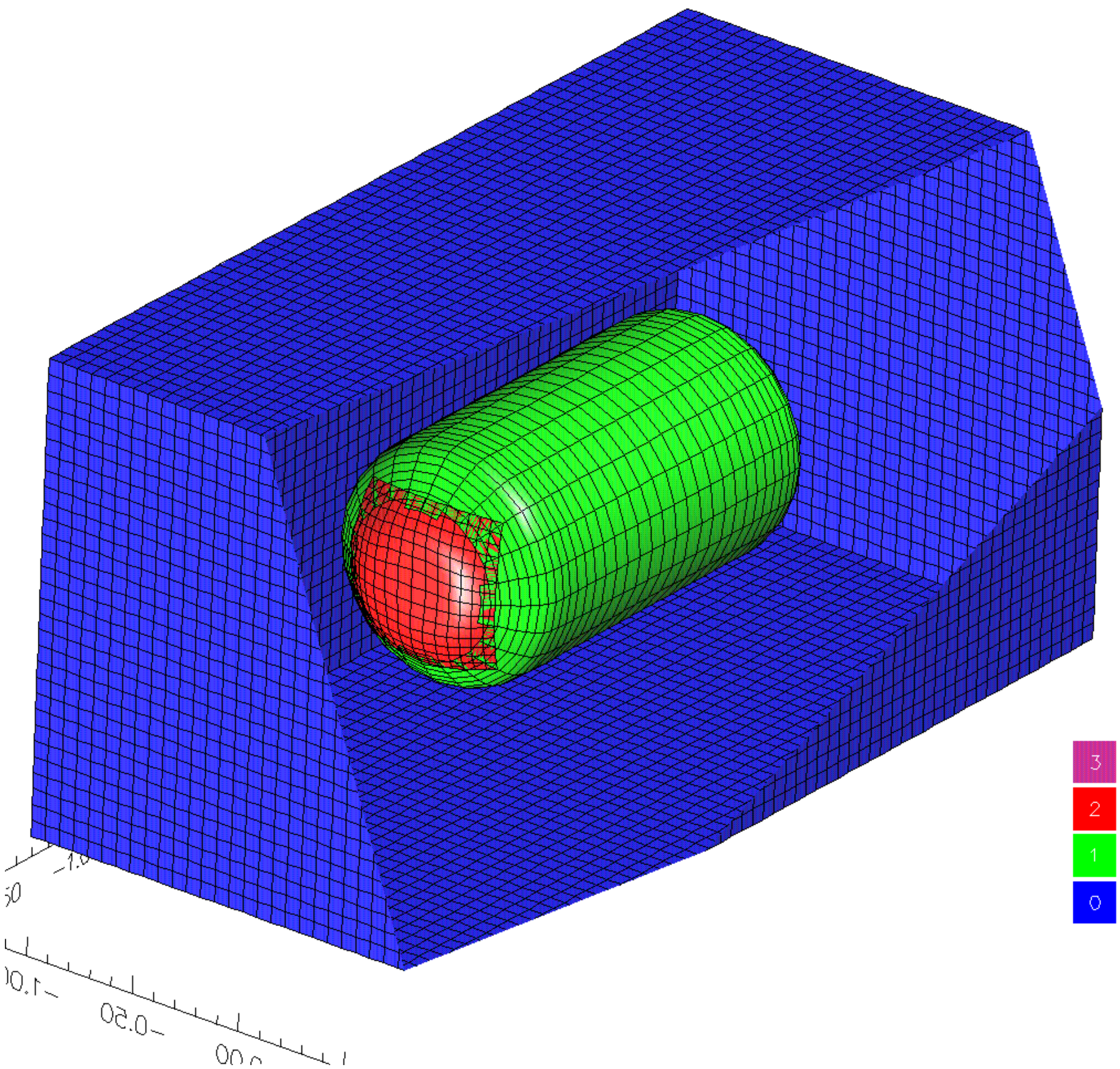


Figure 12: An overlapping grid for a body of revolution. The body is generated by revolving a two-dimensional smoothed-polygon mapping. Orthographic patches are used to cover the singularities at the ends of the body.

4.16 3D valve

Here is a command file to create a grid for a three dimensional valve. The cross-section of this geometry is similar to the two-dimensional valve shown earlier. (file `Overture/sampleGrids/valve3d.cmd`)

```
*
* Make a 3d valve
*
*     cpu=78s (ov15 sun-ultra optimized)
create mappings
  * main cylinder
  Cylinder
    mappingName
      outerCylinder
    * orient the cylinder so y-axis is axial direction
    orientation
      2 0 1
    bounds on the radial variable
      .4 1.
    bounds on the axial variable
      -.1 .5
    lines
      55 11 9
    boundary conditions
      -1 -1 0 3 0 2
    share
      0 0 0 1 0 2
  exit
* core of the main cylinder
  Box
    mappingName
      cylinderCore
    set corners
    -.5 .5  0. .5  -.5 .5
    lines
      19 17 19
    boundary conditions
      0 0 1 2 0 0
    share
      0 0 3 1 0 0
  exit
* valve stem
  Cylinder
    mappingName
      valveStem
    * orient the cylinder so y-axis uis axial direction
    orientation
      2 0 1
    bounds on the radial variable
      .2 .6
    bounds on the axial variable
      -.5 -.2
    lines
      41 9 9
    boundary conditions
      -1 -1 3 2 2 0
    share
      0 0 4 3 0 0
    exit
*  Make a 2d cross-section of the valve
  SmoothedPolygon
    mappingName
      valveCrossSection
    vertices
      4
      .4 0.
      .85 0.
      .65 -.2
      .4 -.2
    n-dist
    fixed normal distance
      .15
    lines
      65 17
    sharpness
      30
      30
      30
      30
    boundary conditions
      0 0 1 0
  exit
*  Make the valve as a body of revolution
  body of revolution
    mappingName
    valve
    choose a point on the line to revolve about
      0. 1. 0.
    lines
      41 11 35
    boundary conditions
      0 0 2 0 -1 -1
    share
      0 0 3 0 0 0
  exit
* 2D cross section for the stopper
  SmoothedPolygon
    mappingName
      stopperCrossSection
    vertices
      4
      .65 -.5
      .65 -.3
      .85 -.1
      1. -.1
    n-dist
      fixed normal distance
      .15
  exit
* stopper
  body of revolution
    mappingName
      stopper
    choose a point on the line to revolve about
      0. 1. 0.
    boundary conditions
      1 1 2 0 -1 -1
    share
      4 2 0 0 0 0
    lines
      35 11 41
  exit
  view mappings
    outerCylinder
    cylinderCore
    valveStem
    valve
    stopper
  exit
exit
*
generate an overlapping grid
    cylinderCore
    outerCylinder
    stopper
    valve
    valveStem
```

```
  done
  change parameters
    ghost points
      all
      2 2 2 2 2 2
  exit
  * pause
  compute overlap
  exit
  save an overlapping grid
  valve3d.hdf
  valve3d
  exit
```

The resulting grid is shown in figure 13.

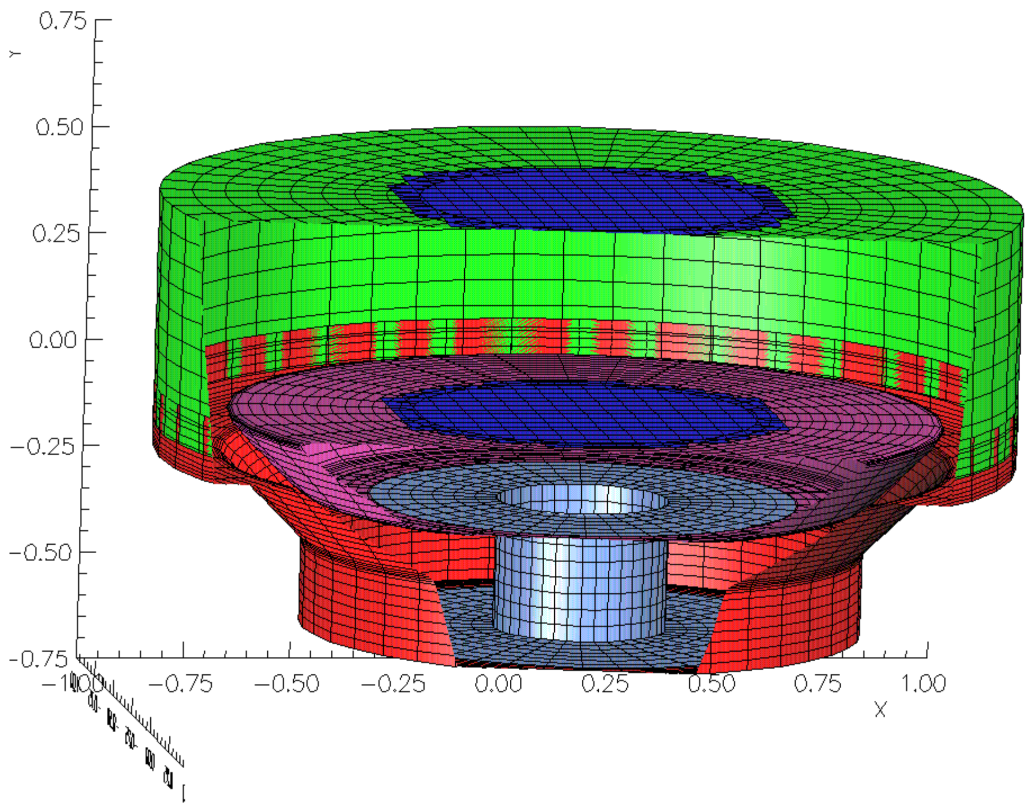


Figure 13: An overlapping grid for a three-dimensional valve.

4.17 Adding new grids to an existing overlapping grid.

New with version 18 This example shows how to start from an existing overlapping grid and add new grids. In this example we begin by building Mappings for two new grids. From the "`generate an overlapping grid`" menu we read in an existing overlapping grid and then specify the additional mappings. Ogen uses an optimized algorithm to compute the new overlapping grid. If for some reason this algorithm fails you can always choose "`reset grid`" followed by "`compute overlap`" to rebuild the grid from scratch.

```
*
* add mappings to an existing overlapping grid
*
create mappings
*
  annulus
  centre
    1. 1.
  boundary conditions
    -1 -1 1 0
  mappingName
    annulus2
  exit
*
  rectangle
    boundary conditions
      0 0 0 0
    set corners
      -1.5 -.5 -1.5 -.5
    mappingName
     refine
    exit
*
  exit this menu
  generate an overlapping grid
    read in an old grid
      cic
    annulus2
    refine
```

The resulting grid is shown in figure 14.

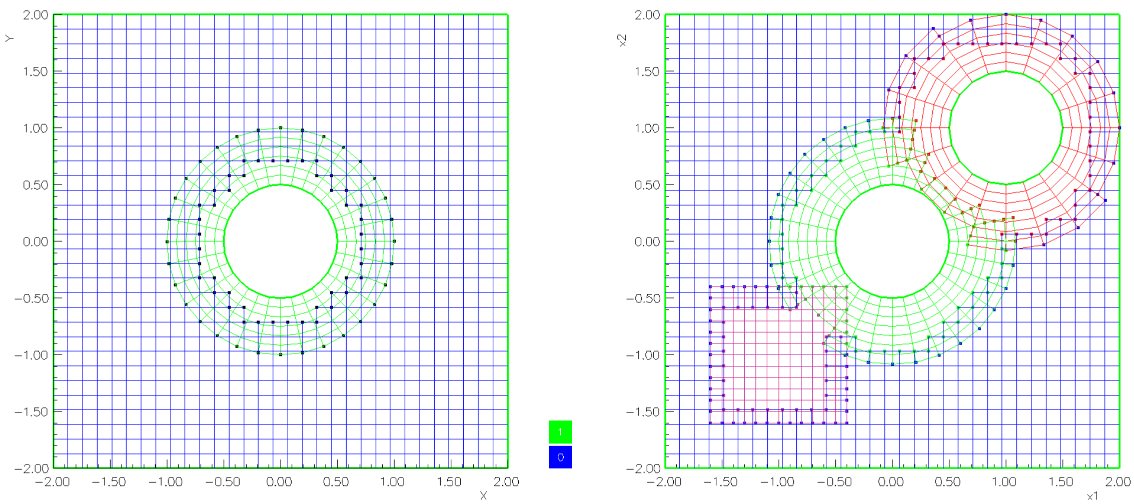


Figure 14: Ogen can be used to incrementally add new grids to an existing overlapping grid. Left: The initial overlapping grid. Right: overlapping grid after adding two new component grids

4.18 Incrementally adding grids to an overlapping grid.

New with version 18 This example shows how to incrementally add new grids to an overlapping grid. As new grids are added the overlapping grid can be re-computed to make sure that a valid grid exists. This can be a useful approach for building a large complicated grid since any problems will be isolated to the component grid that may have caused an invalid grid to result.

```
create mappings
  rectangle
    lines
      41 41
    mappingName
      backGround
    exit
*
  annulus
    inner and outer radii
      .1 .2
    lines
      21 5
    centre for annulus
      .25 .25
    boundary conditions
     -1 -1 1 0
    mappingName
      annulus1
    exit
*
  annulus
    inner and outer radii
      .1 .2
    lines
      21 5
    centre for annulus
      .6 .35
    boundary conditions
     -1 -1 1 0
    mappingName
      annulus2
    exit
*
  annulus
    inner and outer radii
      .1 .2
    lines
      21 5
    centre for annulus
      .35 .65
    boundary conditions
     -1 -1 1 0
    mappingName
      annulus3
    exit
*
  annulus
    inner and outer radii
      .1 .2
    lines
      21 5
    centre for annulus
      .7 .65
    boundary conditions
     -1 -1 1 0
    mappingName
      annulus4
    exit
  exit this menu
  generate an overlapping grid
    backGround
    done choosing mappings
    compute overlap
    pause
    add grids
    annulus1
    done choosing mappings
    compute overlap
    pause
    add grids
    annulus2
    done choosing mappings
    compute overlap
    pause
    add grids
    annulus3
    done choosing mappings
    compute overlap
    pause
    add grids
    annulus4
    done choosing mappings
    compute overlap
```

The resulting grids at various stages are shown in figure 15.

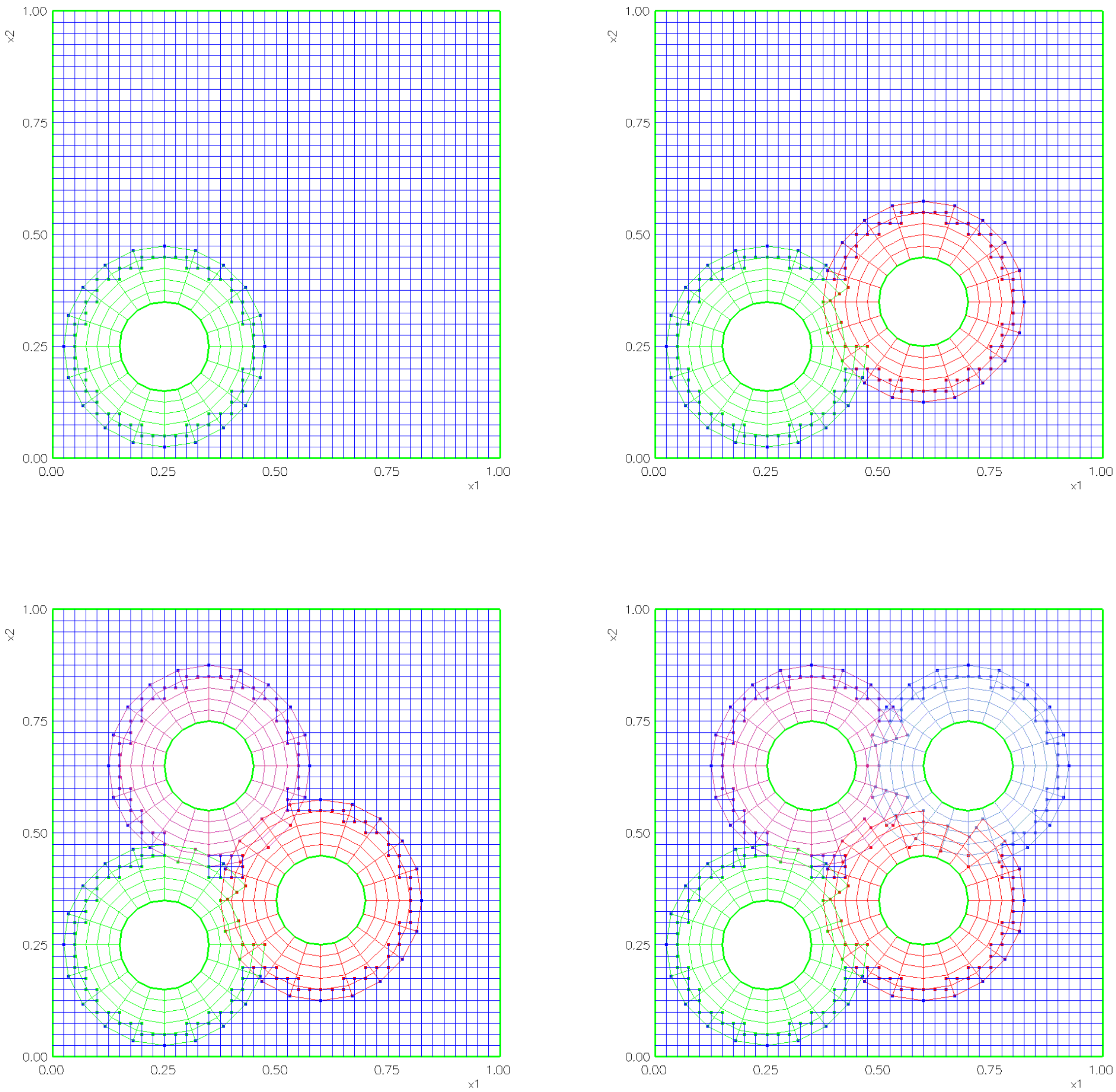


Figure 15: Ogen can be used to incrementally add new grids.

4.19 Other sample command files and grids

The `Overture/sampleGrids` directory contains a number of other command files for creating grids. We list these here with a brief explanation.

cilc.cmd : Two dimensional cylinder in a long box. Used for computing the flow around a cylinder.

ellipsoid.cmd : Create a grid for a three-dimensional ellipsoid in a box. See also `ellipsoidCC.cmd` for the cell-centered version.

singularSphere.cmd : Build a grid for a sphere in a box where the singularities on the sphere are not removed. A PDE solver must know how to deal with this special type of grid.

tse.cmd : Build a grid for a model two-stroke engine.

mastSail2d.cmd : Make a grid for a sail attached to a mast.

building3.cmd : Three dimensional grids for some buildings.

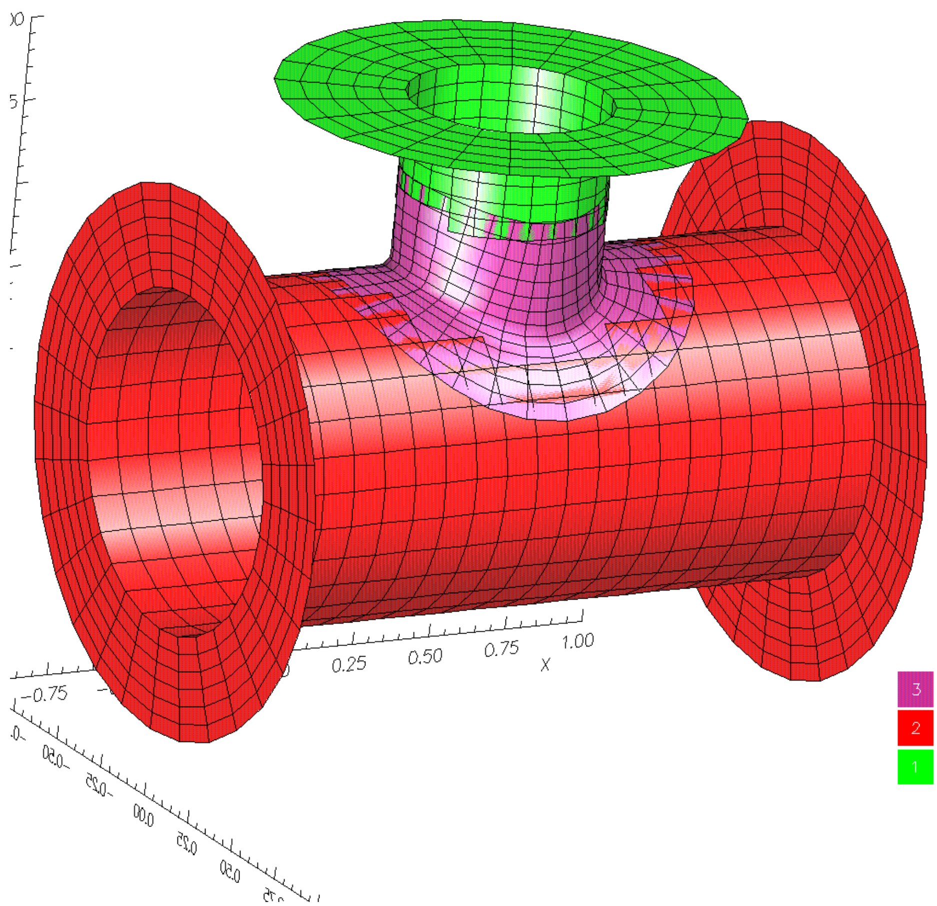


Figure 16: A fillet grid is used to join two cylinders, `filletTwoCyl.cmd`.

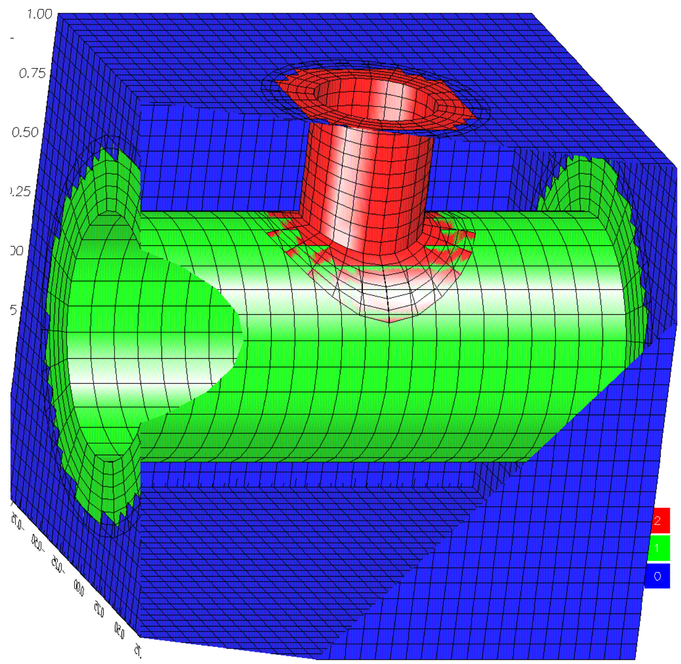


Figure 17: A JoinMapping is used to join two cylinders, `joinTwoCyl.cmd`. To create the deformed cylinder the JoinMapping first computes the curves of intersection between two intersecting cylinders. Four TFIMappings are then generated to represent each face of the deformed cylinder and finally another TFIMapping is used to blend these four surface TFIMappings.

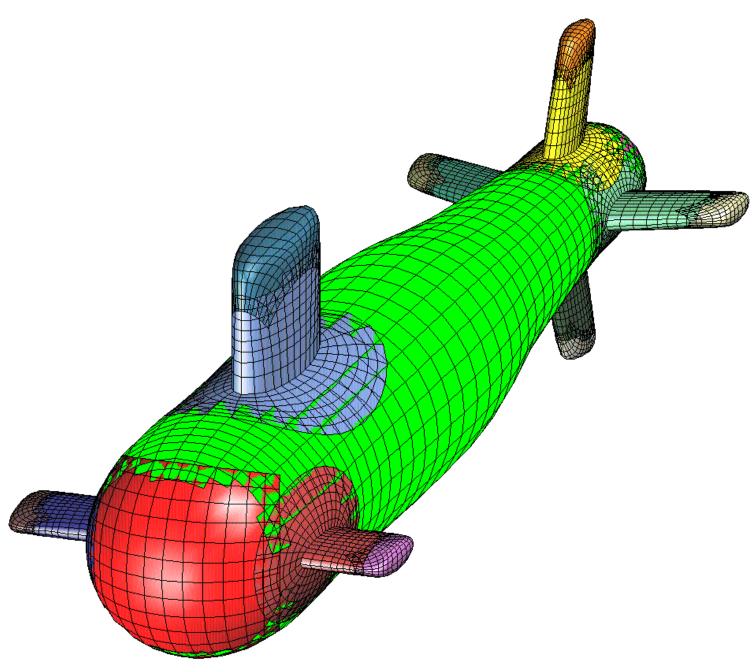


Figure 18: An overlapping grid for a submarine created with `sub.cmd`. The submarine hull is defined as a body of revolution from a spline curve. The sail and fins are created initially with the CrossSectionMapping. The JoinMapping is used to join these appendages to the submarine body.

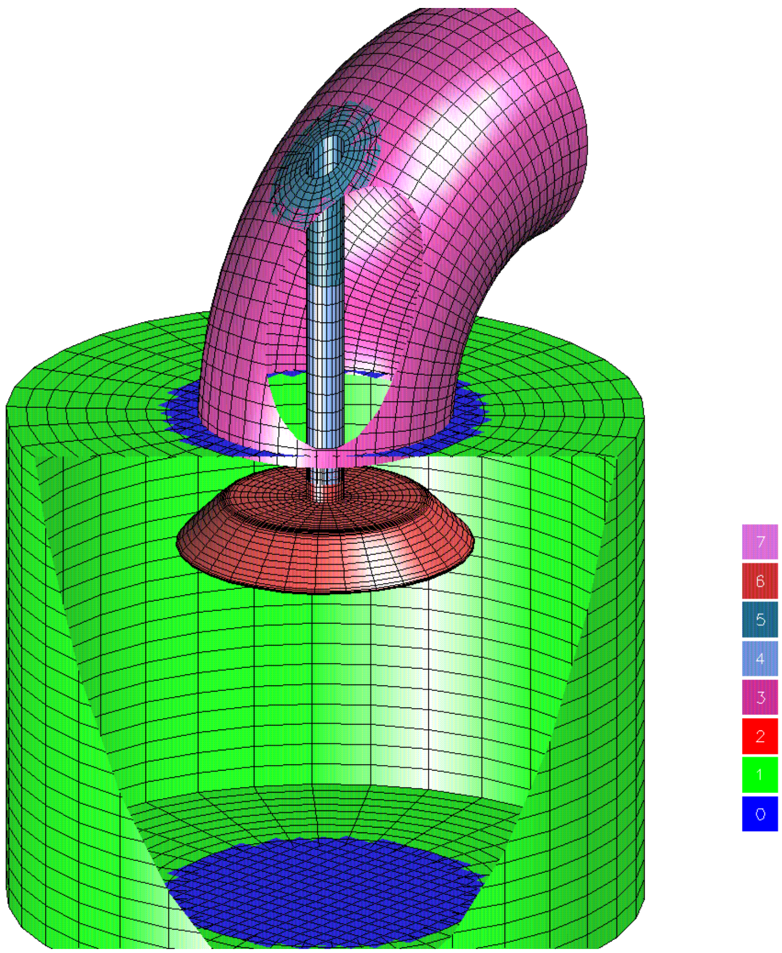


Figure 19: An overlapping grid for valve, port and cylinder created with `valvePort.cmd`. The JoinMapping is used to create the grid that joins the valve-stem to the port surface.

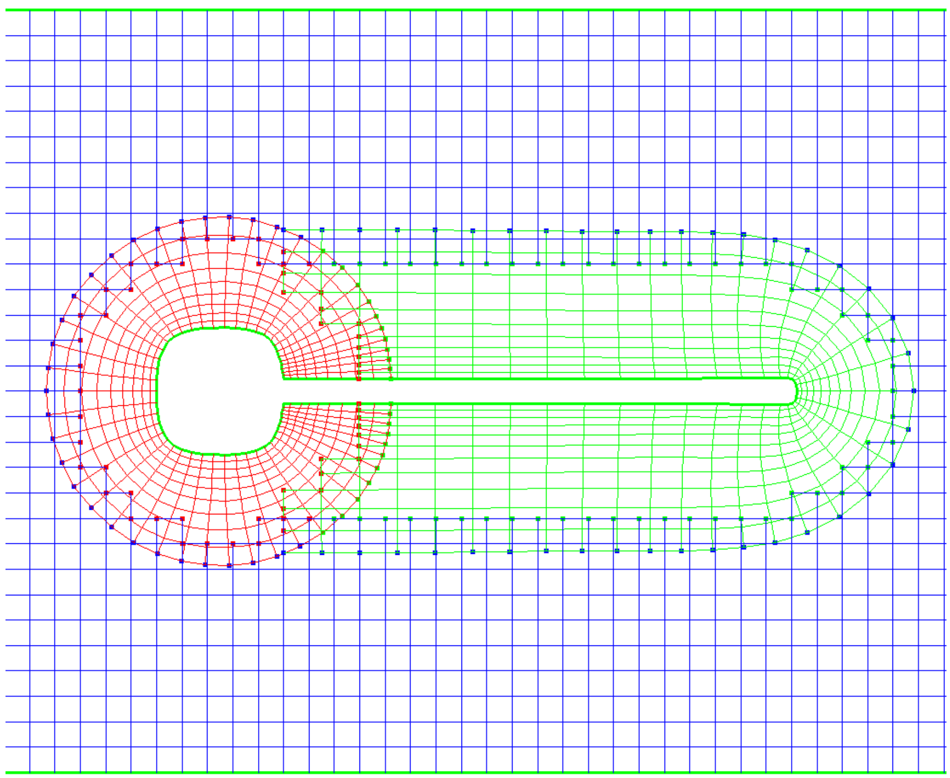


Figure 20: A mast is attached to a sail. The inner boundary curves are defined from splines under tension while the component grids are generated with hyperbolic grid generation `mastSail2d.cmd`

.

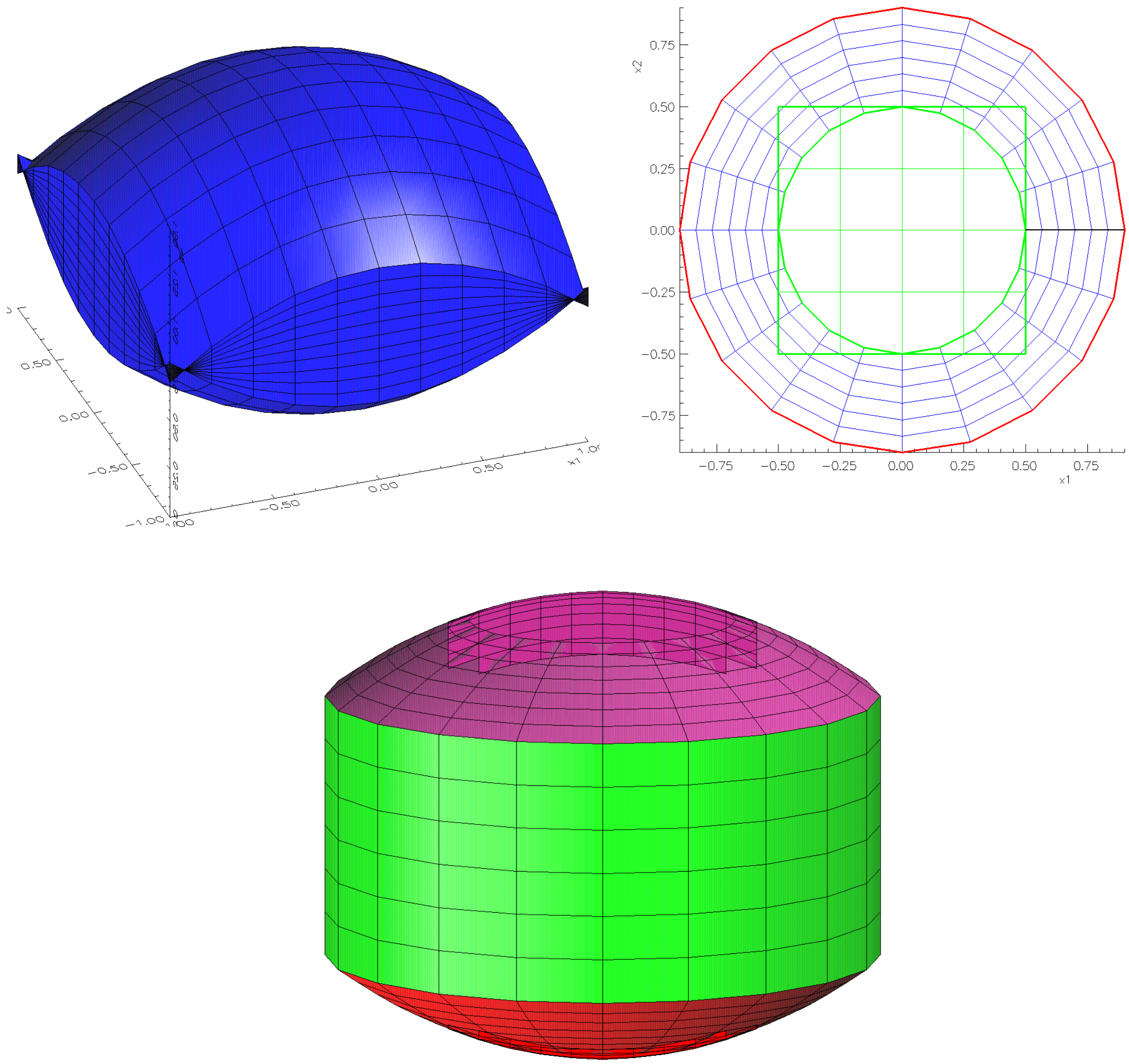


Figure 21: The DepthMapping (see bottom figure) is used to give a vertical dimension to mappings defined in the plane, `depth.cmd`. In this case a separate TFI mapping, top left, defines the vertical height function Both an annulus and a square (top right) are given a depth.

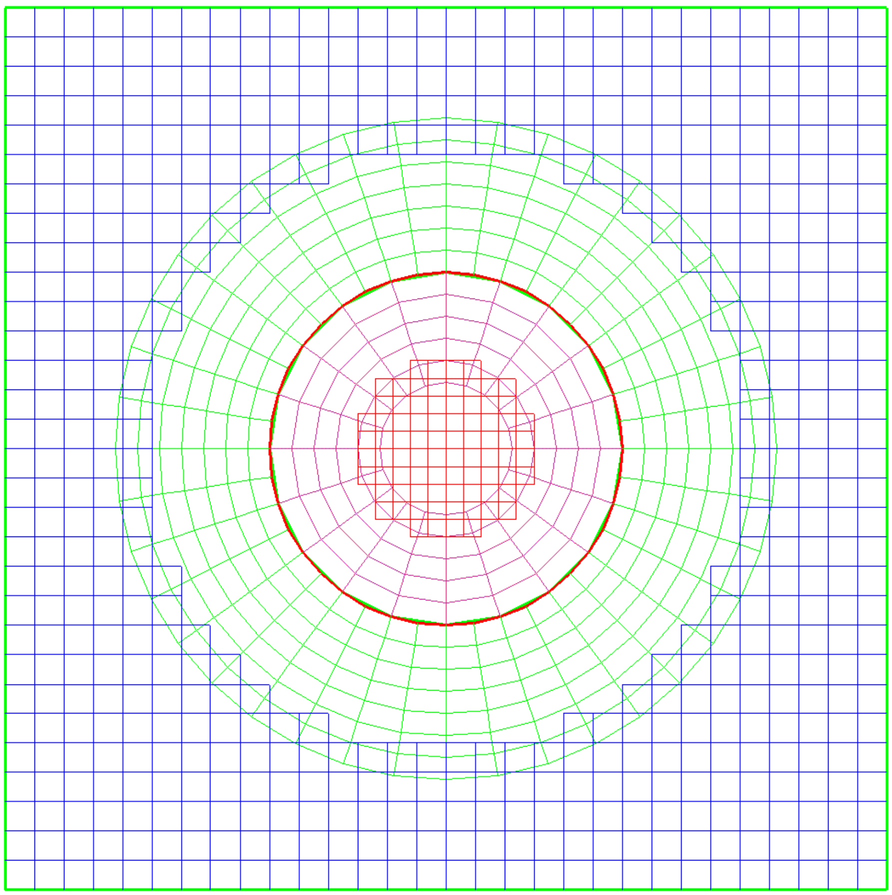


Figure 22: Grids for two disjoint regions that match along a circle, `innerOuter.cmd`

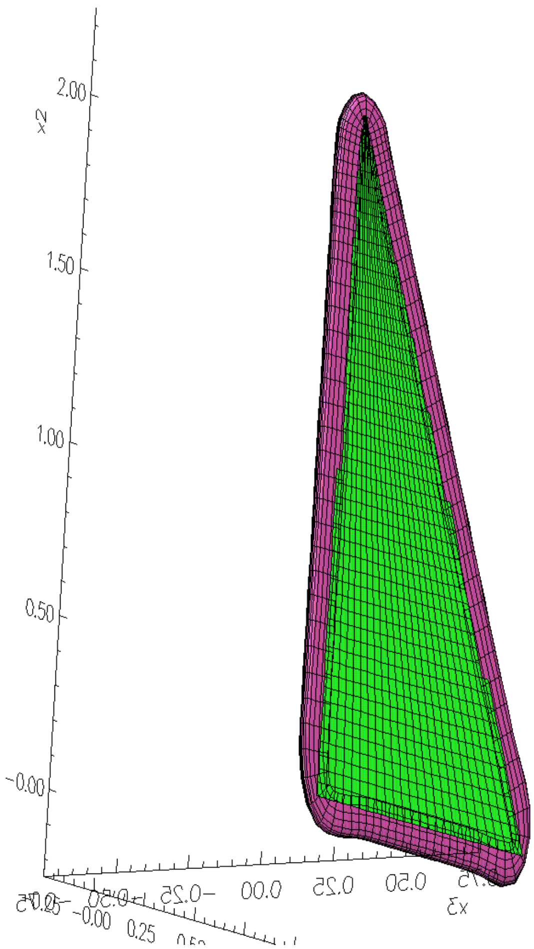


Figure 23: Grid for a 3d triangular sail. The SweepMapping is used to generate a grid around the edge of the sail, `triSail.cmd`

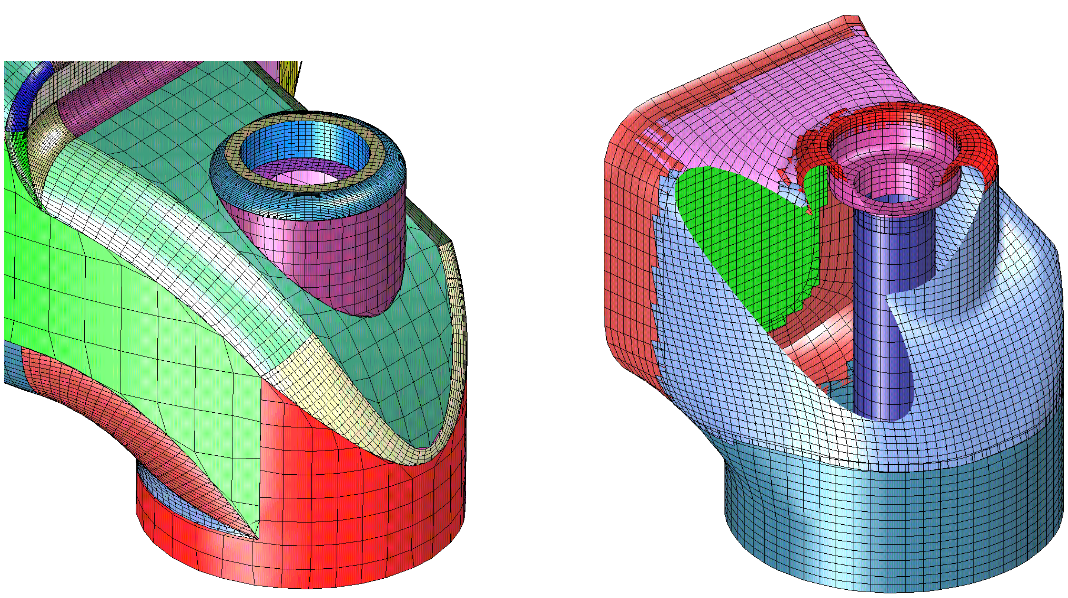


Figure 24: CAD surface (left) and a volume mesh (right) generated with Overture Mappings and Ogen.

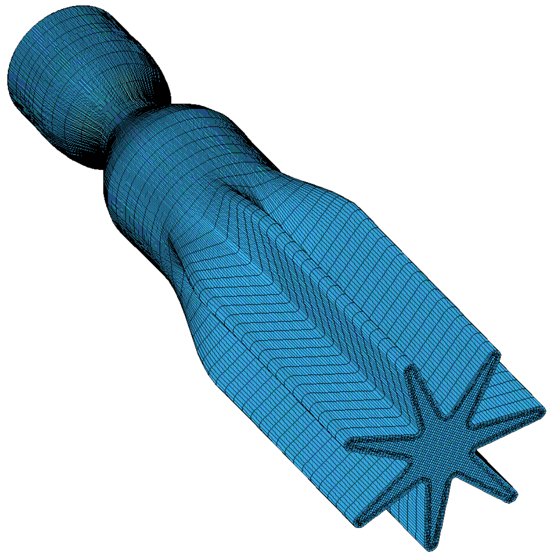


Figure 25: Grid for the core of a rocket, showing the fuel-grain star-pattern. Rocket shape was created with the cross-section mapping and curves defined by the RocketMapping class. Thanks to Nathan Crane for building this grid.

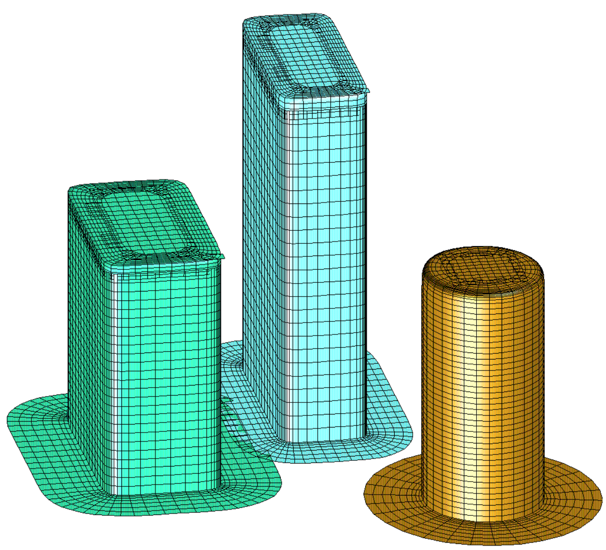


Figure 26: Grid for some buildings built with `building3.cmd`

5 Mixed physical-interpolation boundaries, making a c-grid, h-grid or block-block grid

To make a 'c-grid' as in figure (27) or an 'h-grid' as in figure (28) or the two block grid of figure (29), one should use the 'mixed boundary' option from the change parameters menu. A mixed boundary is a physical boundary where parts of the boundary can interpolate from another (or the same) grid. Actually it is either the boundary points or the ghost points on parts of the boundary that interpolate from another grid. When solving a PDE boundary value problem, the boundary points adjacent to ghost points that interpolate will be 'interior points' where the PDE should be applied, rather than the boundary condition. A mixed boundary on a `MappedGrid g` will have `g.boundaryCondition(side,axis) > 0` and `g.boundaryFlag(side,axis)==MappedGrid::mixedPhysicalInterpolationBoundary`.

There are two ways to determine which points on a mixed boundary should be interpolated

1. **Automatic**: With this option the program will attempt to find all the valid interpolation points. For the automatic determination of the mixed boundary interpolation points you can specify the tolerance for matching in two possible ways:

   r matching tolerance : boundaries match if points are this close in unit square space.

   x matching tolerance : boundaries match if points are this close in x space

   The boundaries will be deemd to match if either one of the above two matching conditions holds.

2. **Manual**: with this option one must explicitly specify a set of points on the boundary that should be interpolated from another grid. One also indicates whether to interpolate boundary points or ghost points. If there are multiple disjoint regions to interpolate, each one should be specified separately. Even when points are specified in this **manual** case the program will still check to see if the points can be interpolated in a valid manner (and only interpolate those valid ones) using the **r matching tolerance** described above.

5.1 Automatic mixed-boundary interpolation

It is recommended when making a c-grid or an h-grid to have the matching parts of the boundaries actually overlap by an amount greater than or equal to zero (as shown in the examples).

The c-grid was generated with the command file `Overture/sampleGrids/cgrid.cmd`. A c-grid has a special topology where parts of the boundary of the c-grid actually become interior points with a periodic like boundary condition. This is implemented in Ogen by the 'mixed boundary' option. Along the c-grid 'branch cut', ghost point values interpolate from the opposite side of the c-grid.

Note that the c-grid boundary was made with a spline that wiggles a little bit along the branch cut. To ensure that the branch cut would be properly found, the lower part of the cut was raised by a small amount so that it would overlap the upper part of the grid (and vice versa to be symmetric). One can also specify a matching tolerance to take care of this problem, but it is more robust to use this trick of overlapping the branch cut a little bit. A matching tolerance was actually specified here, to be safe, but a message printed from ogen indicated that it was not needed. The h-grid was generated with the command file `Overture/sampleGrids/hgrid.cmd`. An h-grid has a special topology where parts of the boundary of the h-grid actually become interior points that match up to a second grid. This is implemented in Ogen by the 'mixed boundary' option. Along the h-grid 'branch cut', ghost point values interpolate from the other grid.

Note that the h-grid boundaries were made with splines that wiggle a little bit along the branch cuts (matching portions). To ensure that the branch cuts would be properly found, the lower part of the cut was raised by a small amount so that it would overlap the upper part of the grid (and vice versa to be symmetric). One can also specify a matching tolerance to take care of this problem, but it is more robust to use this trick of overlapping the branch cut a little bit. A matching tolerance was actually specified here, to be safe, but a message printed from ogen indicated that it was not needed.

The grid in figure (29) was generated with the command file `Overture/sampleGrids/twoBlock.cmd`.

5.2 Manual specification of mixed-boundary interpolation points

The command file `cgrid.manual.cmd` found in the `Overture/sampleGrids` directory shows how to manually create a c-grid by specifying which points should be interpolated. Note that we specify how points on the bottom of the c-grid branch cut interpolate from the top (along the ghost points) and how points on the top boundary interpolate from the bottom.

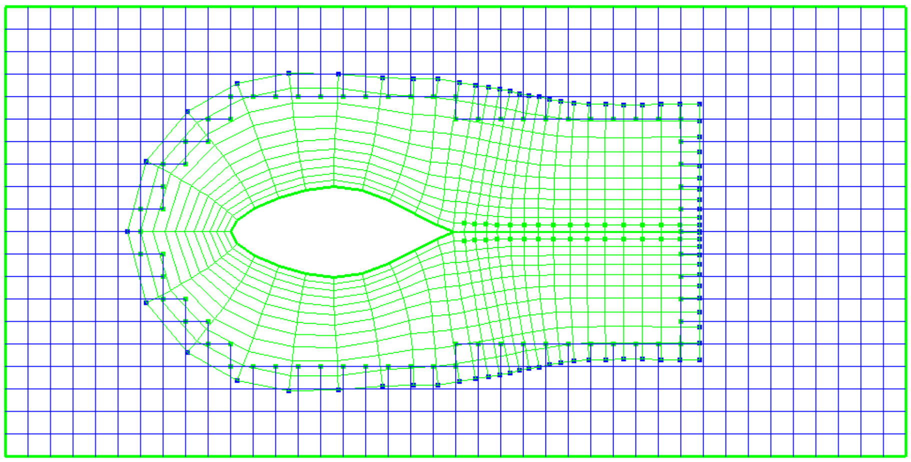


Figure 27: An overlapping grid using a c-grid makes use of the 'mixed boundary' option. A mixed-boundary is a boundary that is sometimes a physical boundary of the domain and sometimes an interpolation boundary.

5.3 Spitting a grid for interpolation of a grid to itself

When mixed boundary interpolation points are to be interpolated from the same grid (as in the case of a c-grid) ogen will actually temporarily split the grid into two pieces and determine how points on one piece interpolate from the other. This is necessary to prevent points from interpolating from themselves. By default, for a mixed boundary on (side,axis) the grid is split at the halfway point along "`(axis+1) mod numberOfDimensions`". If this is not correct you should explicity specify where to split the grid using the `specify split for self interpolation` option. In this case you specify the axis that should be split and the index position of the split.

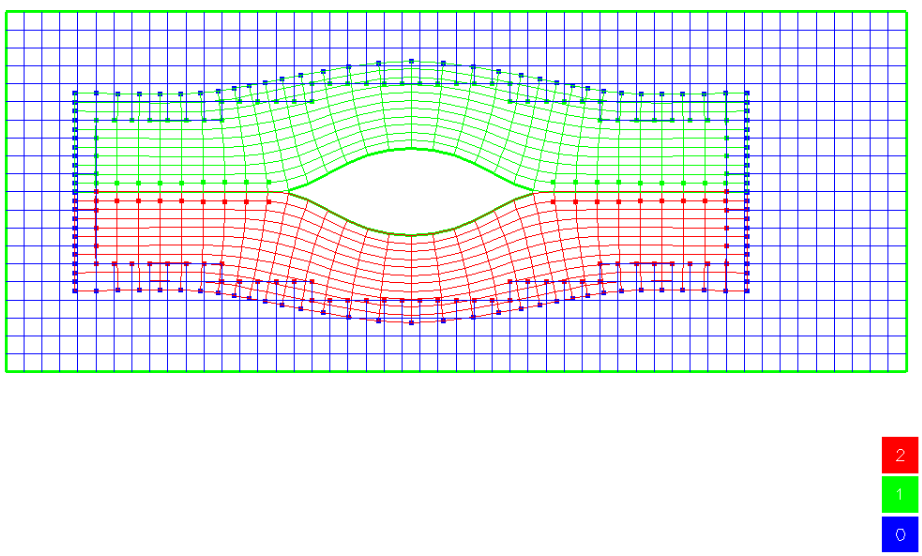


Figure 28: An overlapping grid using an h-grid makes use of the 'mixed boundary' option.

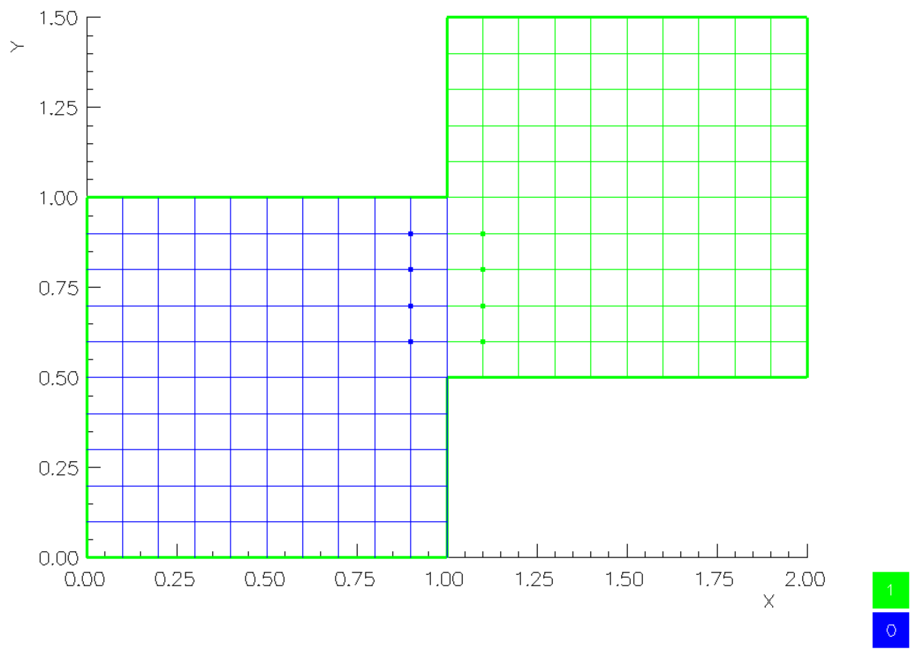


Figure 29: An overlapping grid for two blocks makes use of the 'mixed boundary' option.

6 Manual Hole Cutting and Phantom Hole Cutting

Ogen's hole cutting algorithm can make mistakes in some difficult cases such as when there are thin bodies. There is a *manual hole cutting* option that can be used in these difficult cases. Recall that when ogen cuts a hole with the boundary of grid g_0 it marks points on grid g_1 that lie near the boundary of g_0. Points on g_1 are marked as interpolation or as hole points depending on whether they are inside or outside grid g_0. The hole cutting algorithm can make a mistake if there is a grid g_2 that is very close to the boundary of g_0 but which should not be cut. Normally one can fix this problem by choosing the option *prevent hole cutting* of g_0 in g_2 ; however there are some cases when one must allow g_0 to cut some holes in a different portion of g_2.

There are two steps to perform manual hole cutting:

1. Specify *phantom hole cutting* for grid g_0 onto grid g_1. In this case only interpolation points on g_1 will be marked near the boundary of g_0; no hole points will be marked. These interpolation points should completely surround the hole region.

2. Manually cut a small hole in grid g_1 using the *manual hole cutting* option. The hole points that are specified must lie within the region of g_1 that should be removed. These hole points will act as a seed and will be swept out to fill the entire hole region. If the manually placed hole points are put in the wrong location then the hole points may expand throughout much of the grid, resulting in an invalid overlapping grid.

The command files `cicManualHoleCut.cmd` and `sibManualHoleCut.cmd` in the `Overture/sampleGrids` directory show examples of manually cutting holes.

7 Trouble Shooting

In this section we give some hints on what to do when you are unable to build a grid.

When there is not enough overlap between the grids or you have made a mistake in specifying the boundary conditions or share flag values etc. the grid generator will fail to build a grid. When the algorithm fails the grid will be plotted and the offending points will be plotted with black marks. In addition information is printed to the screen and to a log file, `ogen.log` that may be helpful in tracking down what went wrong.

7.1 Failure of explicit interpolation

As an example, in figures (30) and (31) we show the result of trying to use `explicit interpolation` with the two-dimensional valve grid. The algorithm fails to interpolate some points. These points are plotted with black marks.

Figure 30: An example showing the failure of the overlapping grid algorithm when there is insufficient overlap. We have tried to use explicit interpolation for the two-dimensional valve. The algorithm fails and plots the offending points with black marks.

When the algorithm fails there is information written to the file `ogen.log`. In this case the file contains information on each point that failed, as for example:

```
ERROR: unable to interpolate a point on grid=backGround, (i1,i2,i3)=(26,35,0),  x=( 5.200e-01, 7.000e-01, 0.000e+00)
Try to interpolate from grid=stopper, r=(6.66e-01,5.96e-01,0.00e+00)
```

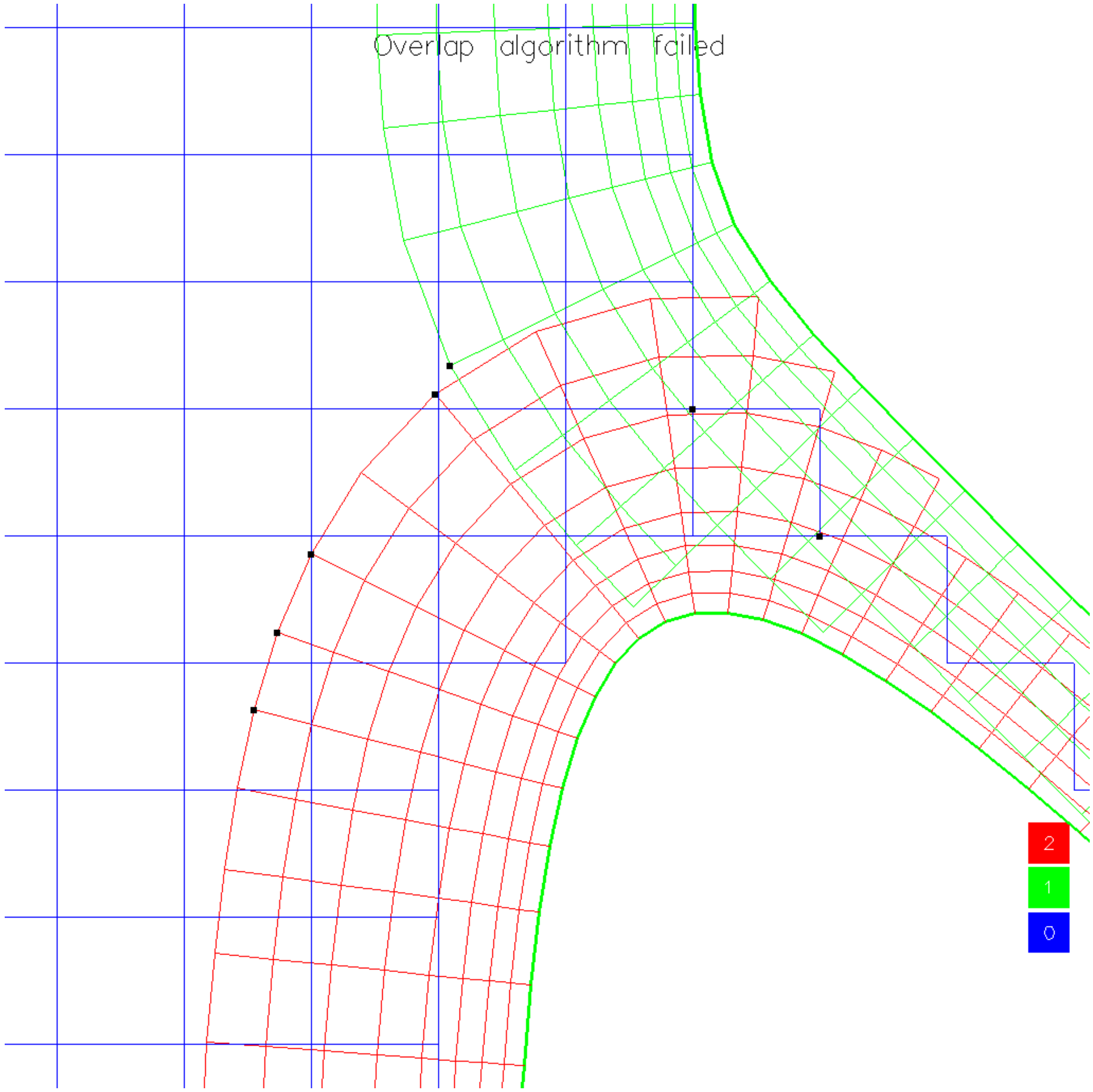


Figure 31: A magnification of the failed grid shows that the points marked in black cannot be interpolated in an explicit manner using a 3×3 interpolation stencil.

```
 mask =[1][1][1][-1][1][1][-1][-1][-1] : 0=hole, -1=interp., 1=discret.
 ...point is inside but explicit interpolation failed because stencil has an interpolation point in it.
Try to interpolate from grid=valve, r=(4.27e-01,4.84e-01,0.00e+00)
 mask =[1][1][1][1][1][1][1][1][-1] : 0=hole, -1=interp., 1=discret.
 ...point is inside but explicit interpolation failed because stencil has an interpolation point in it.
```

This information indicates that a point could not be interpolated from either of two possible grids since the 9-point interpolation stencil (indicated by the 9 values of `mask`) contains some points that are themselves interpolation points (`mask=-1`). The values of `r` indicate the unit square coordinates in the grid we are trying to interpolate from.

Possible solutions to this problem are to use implicit interpolation or to increase the number of grid points on the grids or to decrease the interpolation width.

7.2 Tips

Here are some tips for fixing a grid that fails:

check the log file: Check the ogen log file, **ogen.log** for informational messages that may help you understand what went wrong.

display intermediate results: Turn on the option **'display intermediate results'** in the **ogen** menu before choosing the option **'compute overlap'**. This will plot the grid at intermediate stages in the overlapping grid algorithm.

check the mappings: It is possible that the one of the Mapping's you have created has an error in it. There is a function available to check the properties of a Mapping. The Mapping can be checked either when you create the Mappings (use the `'check mapping'` option) or from the grid generation menu. The checkMapping function will report any errors it finds. For example it will check the derivatives of the mapping by using finite differences. There is probably no reason to be concerned if the relative errors in the derivatives are small, less than $10{-}2$ say.

Use implicit interpolation: As mentioned in section (3.5) implicit (default) interpolation requires less overlap than explicit interpolation. If you are using explicit interpolation you could turn on implicit interpolation.

check boundary conditions: Use the `view mappings` option under `create mappings` to view all the mappings. Check that all physical boundaries are shown as a positive value, that interpolation boundaries have a zero value and that periodic boundaries are black.

check for sufficient overlap: Use the `view mappings` option under `create mappings` to view the mappings and check that the mappings appear to overlap sufficiently. If there is not sufficient overlap then **increase the number of grid points**.

check the share flag: use the `view mappings` option under `create mappings` and plot the boundaries by their share flag value. Make sure that different grids that share the same boundary have the same share flag value (see section (3.2) for a description of share flags).

shared side tolerance: even if your share flags are correct, the grid generator has a relative tolerance that it uses to allow for discrepancies between the boundary representations of two grids. This tolerance measures the distance in grid cells that the boundaries can differ by and still be assumed to be the same boundary. If your boundaries do not match closely then you may need to increase this value with the `shared boundary tolerance` option that is available from the `change parameters` menu.

turn off hole cutting: As described in section (3.3), by default physical boundaries will cut holes in other nearby grids. You may need to disable the hole cutting as shown in the "inlet outlet" example, section (4.7).

8 Adding user defined Mapping's

Advanced users of Overture may want to write their own Mapping class, see the Mapping class documentation for how to do this. If you want to add a new type of Mapping to `ogen` then you should copy and change the driver program `ogenDriver.C` (found in Overture/in) and add in your new Mapping. Compile and load this program to make your own version of ogen.

The next listing shows `ogenDriver.C`. If the preprocessor macro `ADD_USER_MAPPINGS` is defined (for example, by adding the compile flag `-DADD_USER_MAPPINGS` then a user defined `AirfoilMapping` will be added.

```
//=========================================================================================
// Here is the driver program for 'ogen' - the overlapping grid generator
//
//   Usage: type
//       ogen
//  to run with graphics, or type
//       ogen noplot
//   to run without graphics, or
//       ogen file.cmd
//   to run ogen with graphics and read in a command file, or
//       ogen noplot file.cmd
//   to run ogen without graphics and read in a command file.
//
//  By default user commands will be saved in the file "ogen.cmd"
//
//  You can add to the driver any nonstandard Mapping's that you want to use.
//  See the example below where (if the macro ADD_USERMAPPINGS is defined) an AirfoilMapping
//  is created and added to a list. The list is then passed to ogen. The Mapping
//  can be subsequently changed within ogen, if required.
//
//  Thus, for example, your compile line should look something like:
//      CC -DADD_USERMAPPINGS .... ogenDriver.C
//
//=========================================================================================

#include "Overture.h"
#include "MappingInformation.h"
#include "PlotStuff.h"

// Here are some user defined mappings
#ifdef ADD_USER_MAPPINGS
#include "AirfoilMapping.h"
int addToMappingList(Mapping & map);
#endif

int ogen(MappingInformation & mappingInfo, GenericGraphicsInterface & ps, const aString & commandFileName );

int
main(int argc, char *argv[])
{
  Overture::start(argc,argv);
  // Index::setBoundsCheck(off);

  aString commandFileName="";
  if( argc > 1 )
  { // look at arguments for "noplot" or some other name
    aString line;
    for( int i=1; i<argc; i++ )
    {
      line=argv[i];
      if( line=="noplot" || line=="nopause" || line=="abortOnEnd" || line=="nodirect" )
        continue; // these commands are processed by getGraphicsInterface below
      else if( commandFileName=="" )
        commandFileName=line;
    }
  }
  else
    cout << "Usage: 'ogen [noplot][nopause][abortOnEnd][file.cmd]' \n"
            "          noplot:   run without graphics \n"
            "          nopause: do not pause \n"
            "          abortOnEnd: abort if command file ends \n"
            "          file.cmd: read this command file \n";

```

```
  // --- create user defined mappings ----
  MappingInformation mappingInfo;
#ifdef ADD_USER_MAPPINGS
  AirfoilMapping airfoil;
  mappingInfo.mappingList.addElement(airfoil);
  // Do this so we can read the airfoil mapping from a data-base file
  addToMappingList(airfoil);
#endif


  // Graphics interface:
  // Note: options "noplot", "nopause" and "abortOnEnd" are handled in the next call:
  GenericGraphicsInterface & ps = *Overture::getGraphicsInterface("ogen: Overlapping Grid Generator",false,argc,arg

  // By default start saving the command file called "ogen.cmd"
  aString logFile="ogen.cmd";
  ps.saveCommandFile(logFile);
  cout << "User commands are being saved in the file `" << (const char *)logFile << "'\n";

  // create more mappings and/or make an overlapping grid
  ogen( mappingInfo,ps,commandFileName);

  Overture::finish();
  return 0;
}
```

9 Overlapping Grid Generator: Ogen

The overlapping grid generation algorithm determines how the different component grids communicate with each other. The algorithm must also determine those parts of component grids that are removed from the computation because that part of the grid either lies underneath another grid of higher priority or else that part of the grid lies outside the domain.

9.1 Command descriptions

9.1.1 Interactive updateOverlap

int
updateOverlap(CompositeGrid & cg, MappingInformation & mapInfo)

Description: Here is a description of some of the commands that are available from the `updateOverlap` function of `Ogen`. This function is called when you choose "`generate overlapping grid`" from the `ogen` program.

compute overlap : this will compute the overlapping grid. As the grid is generated various information messages are printed out. Some of these messages may only make sense to the joker who wrote this code.

change parameters : make changes to parameters. See the next section for details.

display intermediate results : this will toggle a debugging mode. When this mode is on, and you choose `compute overlap` to generate the grid, then the overlapping grid will be plotted at various stages in its algorithm. The algorithm is described in section (9.2). The program will pause at the end of each stage of the algorithm and allow you to either `continue` or to `change the plot` as described next. Experienced users will be able to see when something goes wrong and hopefully detect the cause.

change the plot : this will cause the grid to be re-plotted. You will be in the grid plotter menu and you can make changes to the style of the plot (toggle grids on and off, plot interpolation points etc.). These changes will be retained when you exit back to the grid generator.

9.1.2 Non-interactive updateOverlap

int
updateOverlap(CompositeGrid & cg)

Description: Build a composite grid non-interactively using the component grids found in cg. This function might be called if one or more grids have changed.

Return value: 0=success, otherwise the number of errors encountered.

9.1.3 Moving Grid updateOverlap

int
updateOverlap(CompositeGrid & cg,
CompositeGrid & cgOld,
const LogicalArray & hasMoved,
const MovingGridOption & option =useOptimalAlgorithm)

Description: Determine an overlapping grid when one or more grids has moved. **NOTE:** If the number of grid points changes then you should use the `useFullAlgorithm` option.

cg (input) : grid to update

cgOld (input) : for grids that have not moved, share data with this CompositeGrid.

hasMoved (input): specify which grids have moved with hasMoved(grid)=TRUE

option (input) : An option from one of:

```
enum MovingGridOption
{
  useOptimalAlgorithm=0,
  minimizeOverlap=1,
  useFullAlgorithm
};
```

The `useOptimalAlgorithm` may result in the overlap increasing as the grid is moved.

Return value: 0=succuss, otherwise the number of errors encountered.

hangeParametersInclude.tex

9.2 Algorithm

The algorithm used by Ogen is based upon the original CMPGRD algorithm[1] with some major changes to improve robustness. The basic improvement is that the new algorithm initially removes all grid points that lie inside "holes" in the grids. Once the holes have been cut the program can determine explicitly whether there is enough overlap to generate an overlapping grid and if there is not enough overlap the offending points can be shown.

The algorithm for computing the overlapping grid communication is perhaps most easily understood by reading the following description and also referring to the series of examples that follow.

Here are the basic steps in brief:

interpolate boundaries: First try to interpolate points on physical boundaries from points on physical boundaries of other grids.

Boundary points that interpolate from the interior of other grids are marked either as being an `interiorBoundaryPoint` and an `interpolationPoint` (using a bitwise 'or' in the mask).

mark hole boundaries: For each physical boundary find points on other grids that are near to and inside or outside of the boundary. After this step the holes in the grid will be bounded by a boundary of holes points next to a boundary of interpolation points.

remove exterior points: Mark all remaining hole points. These points can be easily swept out since the hole cutting algorithm ensures that all holes are bounded by interpolation points.

classify (improper) interpolation boundary: The points on the stairstep boundaries and interpolation boundaries are collected into a list. We first try to interpolate these points from other grids using improper interpolation. A point is said to interpolate in an improper way from a grid if it simply lies within the grid. Since all the points in the list lie within in the domain they must interpolate from some other grid or else there is something wrong. See the section on trouble-shooting for examples when this step fails.

classify proper interpolation boundary: We now take the list of (improperly) interpolated points and sort them into one of the following categories:

proper interpolation: A point of a grid interapolates in a proper way from a second grid if the appropriate stencil of points exists on the second grid and consists of the correct types of points for the implicit or explicit interpolation.

discretization point: An interpolation point on a physical boundary may be used as a dicretization point.

At the successful completion of this step we should have a valid overlapping grid. There should be no fatal errors in performing the final steps.

interpolate discretization points: To reduce the amount of overlap we attempt to interpolate discretization points from grids of higher priority.

remove redundant interpolation points: Any interpolation points that are not needed are removed from the computation. Interpolation points that are needed but that can just as well be used as discretization points are turned into discretization points.

9.3 Hole cutting algorithm

After checking for interpolation points on boundaries, the next step in the overlapping grid algorithm is to cut holes. This is the most critical step in the algorithm. Each side of a grid that represents a physical boundary is used to cut holes in other grids that overlay the boundary.

Each face on grid g representing a physical boundary is used to cut holes in other grids. We also mark points that can interpolate from grid g. The goal is to build a barrier of hole points next to interpolation points that partitions the grid into two regions – one region that is inside the domain and one region that is outside the domain.

- We check for points, $\mathbf{x}_g$ on the face of grid g that can interpolate from from another grid g_2. These points $\mathbf{i}_2$ on g_2 are potential hole points.
- A potential hole point is not cut if it can interpolate from grid g, in this case the point is marked as an interpolation point.

- A potential hole point is NOT cut if the distance to the cutting surface is greater than $2\Delta x_2$ where Δx is a measure of the cell size on g_2 (currently the length of the diagonal of the cell $\mathbf{i}_2$). Thus in general there will be a layer of 1-3 points cut near the cutting surface.
- A potential hole point is NOT cut if the point $\mathbf{i}_2$ already can interpolate from another grid g_3 AND the grid g_3 shares the same boundary with grid g. This condition applies to a thin body and prevents points from being cut that are actually inside the domain on the opposite side of the thin body.

This section needs to be completed...

1. Invert the points $\mathbf{x}_g$ on grid g_2 given coordinates $\mathbf{r}_{g_2}$.
2. Compute the `holeMask` mask array which indicates whether a point on the cutting face is inside of outside g_2

```
Compute the holeMask:
    holeMask(i1,i2,i3) = 0 : point is outside and not invertible
                       = 1 : point is inside
                       = 2 : point is outside but invertible

                     ------------------------
                     |                      |
                     |       grid2          |
   holeMask          |                      |
     --0---0---2---2---1---1---1---1---1---1---2---2---2---0---0---- cutting curve, grid
                     |                      |
                     |                      |
                     |                      |
                     |                      |
                     |                      |
                     ------------------------
```

3. The idea now is to mark all points on g_2 that are near the cutting face.

9.4 Finding exterior points by ray tracing

*** Ray tracing is NO longer performed to remove holes points*** but it is used to generate embedded boundary grids (a future feature).

Exterior points are found by counting the number of times that a semi-infinite ray, starting from a point $\mathbf{x}$ and extending in the y-direction to $+\infty$, crosses the boundaries of the region. If the ray crosses the boundaries an even number of times then it is outside the domain.

If a ray crosses the region where two grids overlap then there will appear to be two points of crossing. We must eliminate one of these points of crossing or else we will obtain an incorrect result.

The ray casting algorithm will determine the intersection of the ray with the boundary surfaces represented as a triangulation of the discrete points.

We keep a list of the positions of intersection, $\mathbf{x}_i$, as well as the grid and grid point location of the intersection. Ideally we would only need to check whether two points of intersection from two different grids are close, $\|\mathbf{x}_i - \mathbf{x}_j\| < \epsilon$. It is not very easy, however, to determine an appropriate value for ϵ. If the ray crosses the boundary in a nearly normal direction then the distance $d = \|\mathbf{x}_i - \mathbf{x}_j\|$ will be of order the discrepency between the two discrete representations of the surface which can be estimated by ??

If, however, the ray crosses the boundary in a nearly tangential direction then the distance d could be as large as the grid spacing in the tangential direction.

There are further complications since the body may represent a very thin surface (such as a wing) and there may be points of intersection that are close together in physical space but actually on opposite sides of the thing body.

Thus to perform a robust check we do the following

1. Check that two intersecting points belong to two different grids, $g_1 \neq g_2$.
2. Check that the boundaries on the two grids are shared sides (meaning they belong to the same surface as specified in the grid generation by setting the `share` flag).

3. Check that the grid cells that contain the points of intersection have some vertices that are interpolation points (so that we know we are in a region of overlap) ???

4. check that the normals to the boundary at the points of intersection point in the same basic direction, $\mathbf{n}_1 \cdot \mathbf{n}_2 > 0$.

5. check that the distance $d = \|\mathbf{x}_i - \mathbf{x}_j\|$ between the points satsifies

$$\begin{aligned} \alpha &= |(\mathbf{x}_2 - \mathbf{x}_1) \cdot \mathbf{n}| / ||(\mathbf{x}_2 - \mathbf{x}_1)|| \qquad 0 \le \alpha \le 1 \\ d_n &\equiv \text{ normal discrepency} \\ d_t &\equiv \text{ tangential discrepency} \\ d &\le \alpha d_n + (1-\alpha) d_t \end{aligned}$$

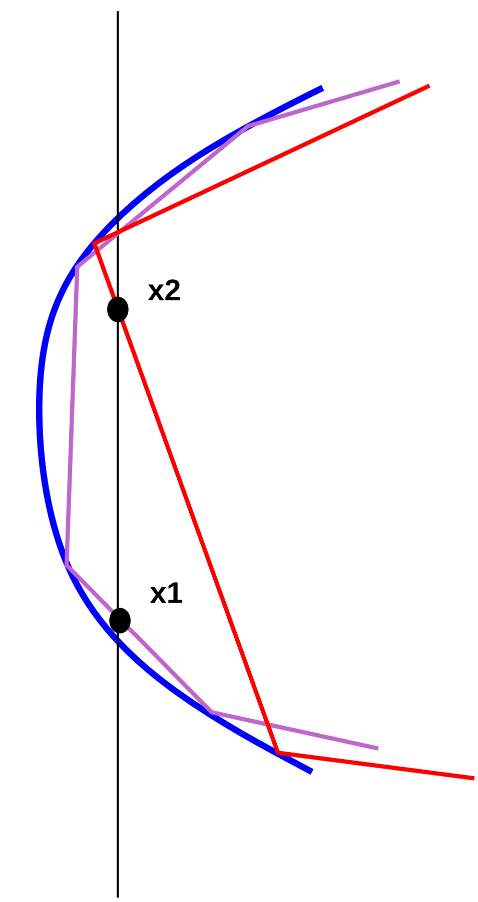


Figure 32: The points of intersection of a ray with a surface covered by two overlapping grids. If the ray is nearly tangent to the surface then the two points of intersection may not be very close together.

9.5 Adjusting grid points for the boundary mismatch problem

When the sides of two grids overlap on a boundary then there can be a problem interpolating one grid from the other if the grids do not match well enough. This problem is especially likely if the grids are formed by interpolating data points and the grid spacing is highly stretched in the normal direction.

Figure (33) shows two grids that share a boundary. If we suppose that the mapping for the grid is defined by linear interpolation between the grid points then it is clear that points on the boundary of grid A appear to be well outside or well inside the boundary of grid B, when actually the boundaries are meant to be the same.

This *boundary mis-match* causes two problems. The first problem, encountered by the grid generator, is that those boundary points (or even interior points for highly strteched grids) that appear to be outside the grid should actually be allowed to interpolate. The hole cutting algorithm will mark these points as being unusable and outside the grid. The second problem occurs in PDE solvers. Even if we allow the points to interpolate, the interpolation will not be very accurate and the solution can look bad.

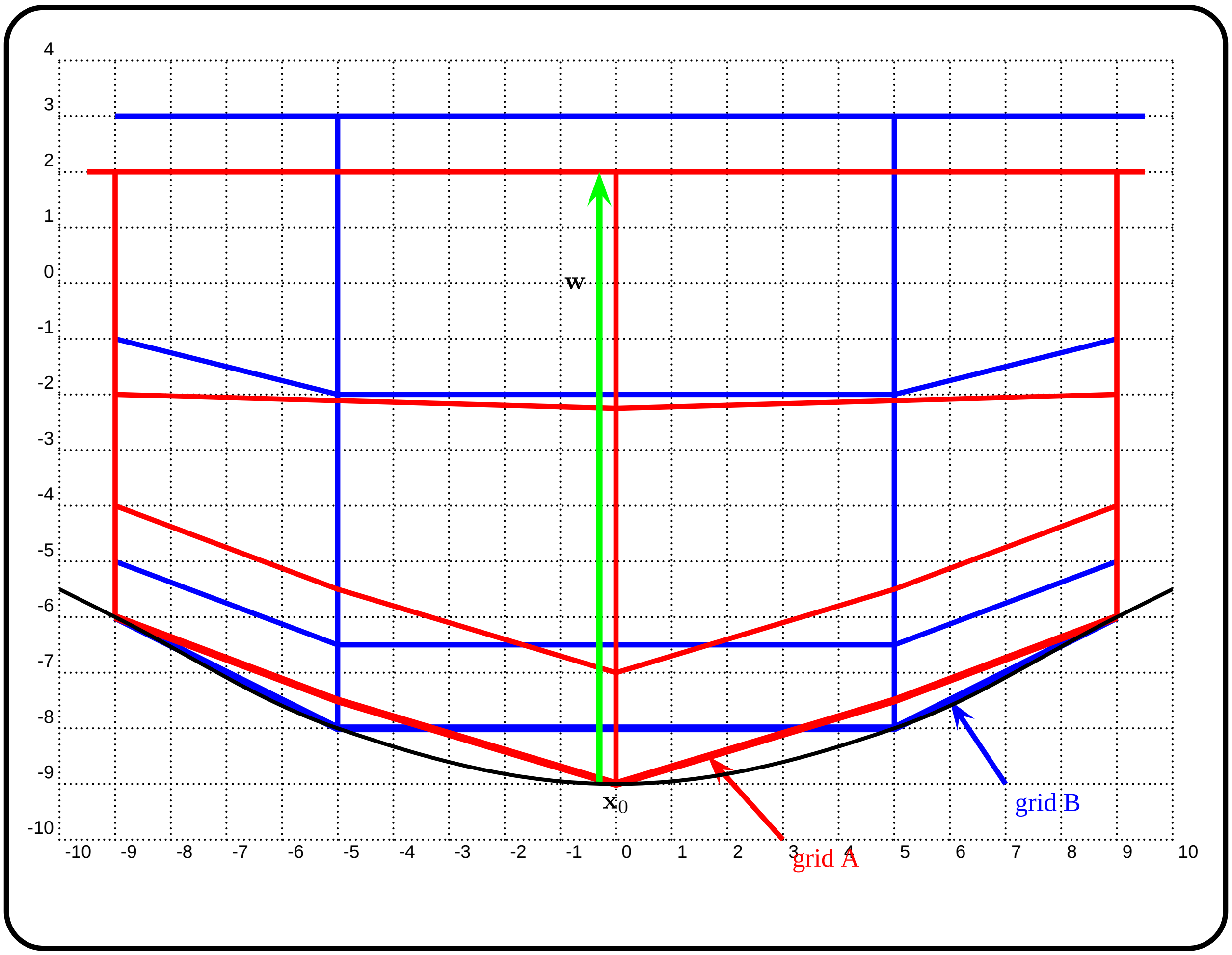


Figure 33: Grid A and Grid B share a boundary but if the mappings are defined by linear interpolation, the grid point $\mathbf{x}_0$ will appear to be outside grid B.

To fix both these problems we adjust the points on grid A so that the boundary points of grid A are shifted to lie exactly on the boundary of grid B. Other points on grid A are also shifted, but the amount of the shift decreases the further we are from the boundary. If the grid is highly stretched then the relative amount we shift the points, compared to the local grid spacing, decreases as we move away from the boundary. For example if the spacing near the boundary is 10^{-3} compared to the spacing away from the boundary layer then the amount we shift interior points will be on the order of 10^{-3}, a very small relative change. **Note that this shift is only done when we are determining the location of A grid points in the parameter space of grid B (for interpolation)**. The actual grid points are not changed in the `CompositeGrid` created by the grid generator. Also note that points on grid A may be shifted one amount when interpolating from grid B, but could be shifted another amount if interpolating from a third grid C.

Referring to figure (34) the point $\mathbf{x}_0$ is shifted to the point $\mathbf{x}_1$ on the boundary. The point $\mathbf{x}_2$ is also shifted, but by a smaller amount, that depends on the distance from the boundary relative to the vector $\mathbf{w}$

$$\begin{aligned}
\tilde{\mathbf{x}_2} &\leftarrow \mathbf{x}_2 + (\mathbf{x}_1 - \mathbf{x}_0)[1 - \frac{(\mathbf{x}_2 - \mathbf{x}_0)\cdot\mathbf{w}}{\|\mathbf{w}\|^2}] \\
&\equiv \mathbf{x}_2 + (\mathbf{x}_1 - \mathbf{x}_0)[1 - \frac{(\mathbf{x}_2 - \mathbf{x}_0)\cdot\mathbf{w}}{\|\mathbf{w}\|^2}] \\
&\equiv \mathbf{S}(\mathbf{x}_1)\mathbf{x}_2
\end{aligned}$$

The *opposite-boundary* vector $\mathbf{w}$ is chosen to extend from the boundary to the grid points as some distance from the boundary. We use the grid line that is at least 1/3 of the distance (in index space) to the opposite side, but at least 10 lines (unless there are fewer than 10 lines). The vector should be far enough away so that points in the boundary layer are shifted to be inside the other grid, but close enough so that $\mathbf{w}$ is nearly parallel to the normal to the boundary.

The shift operator $\mathbf{S}$ will project the boundary points of grid A onto the boundary of grid B.

A complication occurs if the more than one side of grid A shares sides with the same grid B, as shown in figure (34). In this case we must determine shifts in multiple directions so that after these shifts the boundary points on grid A are shifted to lie on the boundary of grid B. We cannot simply apply the above algorithm for each side independently.

To fix this problem we sequentially apply the shift operations more than once in order to ensure that the grids points are projected onto all the shared boundaries. Let $\mathbf{S}_0$, $\mathbf{S}_1$ and $\mathbf{S}_2$ denote the shift mappings in each coordinate direction. In two dimensions, the operator

$$\tilde{\mathbf{x}_2} \leftarrow \mathbf{S}_1\mathbf{S}_0\mathbf{x}$$

will not work properly since after the application of $\mathbf{S}_1$ the points on boundary 0 can be shifted off the boundary. However the operator

$$\tilde{\mathbf{x}_2} \leftarrow \mathbf{S}_0\mathbf{S}_1\mathbf{S}_0\mathbf{x}$$

would work since the final $\mathbf{S}_0$ operator will not change the points on boundary 1 (since the corner points of grid A have been projected to the corner points of grid B after the two steps $\mathbf{S}_1\mathbf{S}_0\mathbf{x}$).

Rather than applying $\mathbf{S}_0$ twice it is more efficient to define new operators to perform the projection in only two steps:

$$\tilde{\mathbf{x}_2} \leftarrow \widetilde{\mathbf{S}}_1\widetilde{\mathbf{S}}_0\mathbf{x}$$

We can do this

$$\begin{aligned}
\widetilde{\mathbf{S}}_0 &= \mathbf{S}_0(\mathbf{x}_1 + \mathbf{y}) \\
\mathbf{y} &= \mathbf{S}_0(\mathbf{x}_1)\mathbf{x}_1 \\
\widetilde{\mathbf{S}}_1 &= \mathbf{S}_1
\end{aligned}$$

In three-dimensions if we have three adjacent shared faces then

$$\begin{aligned}
\tilde{\mathbf{x}_2} &\leftarrow \widetilde{\mathbf{S}}_2\widetilde{\mathbf{S}}_1\widetilde{\mathbf{S}}_0\mathbf{x} \\
\widetilde{\mathbf{S}}_0 &= \mathbf{S}_0\mathbf{S}_2\mathbf{S}_1\mathbf{S}_0 \\
\widetilde{\mathbf{S}}_1 &= \mathbf{S}_1 \\
\widetilde{\mathbf{S}}_1 &= \mathbf{S}_2
\end{aligned}$$

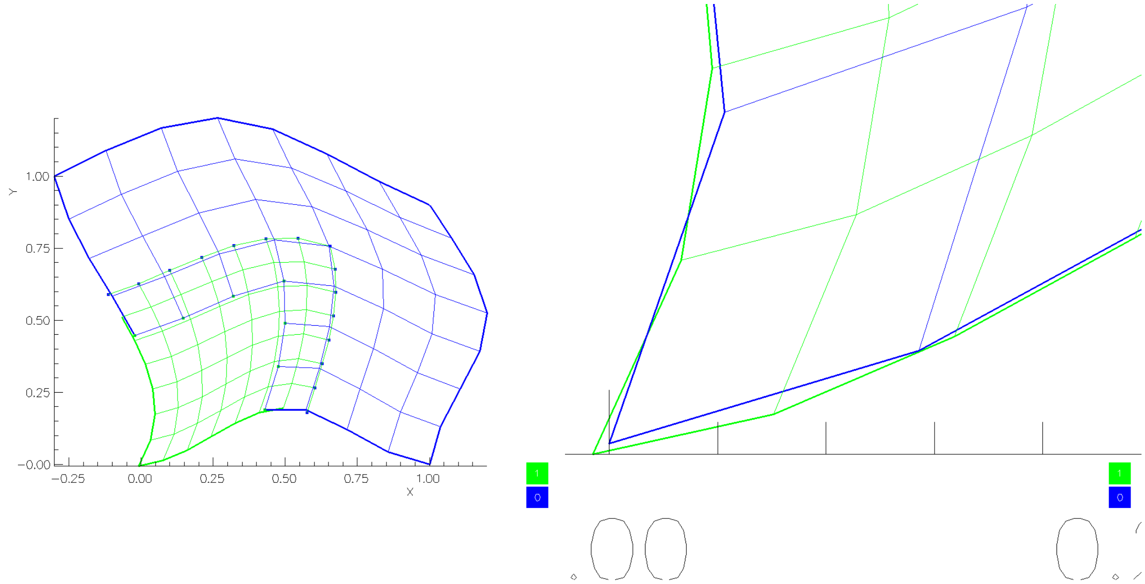


Figure 34: An overlapping grid testing the mismatch problem, created with `mismatch.cmd`. The refinement grid is artifically translated so that the two boundaries it shares with the base grid do not match. The figure on the right is a magnification of the lower left corner, before the overlap algorithm was applied.

9.6 Refinement Grids

Refinement grids can be added to a `GridCollection` or to a `CompositeGrid`. The component grids that exist in the original `CompositeGrid`are known as **base grids**. These grids represent **refinement level 0**. Refinement grids are added on a particular base grid and belong to a particular level. Normally the refinement levels are **properly nested** so that all grids on refinement level l are contained in the grids on refinement level $l-1$.

A given refinement grid will have only one parent grid on refinement level 0, i.e. it will belong to only one base grid. A refinement grid on level l may have more than one parent grid on level $l-1$.

Normally a refinement grid will interpolate its ghost values from other refinement grids on the same level or from its parent grids. Points on the parent grid that lie underneath the refinement will interpolate from the refinement (also known as the child grid).

If refinement grids lie in a region where two base grids overlap, it is necessary to determine how the refinements interpolate from the grids they overlap that belong to a different base grid.

The `updateRefinements` function determines how refinement grids interpolate from other grids that they overlap. This function does not determine how a refinement grid interpolates from the grid it has refined.

If a refinement...

9.7 Improved Quality Interpolation

This is new* Version 16 or higher.

Normally one wants to avoid having a fine grid interpolate from a coarse grid or vice versa. Often this can be accomplished through the normal specification of a priority for each grid. Sometimes, however, using a single priority per grid is not sufficient.

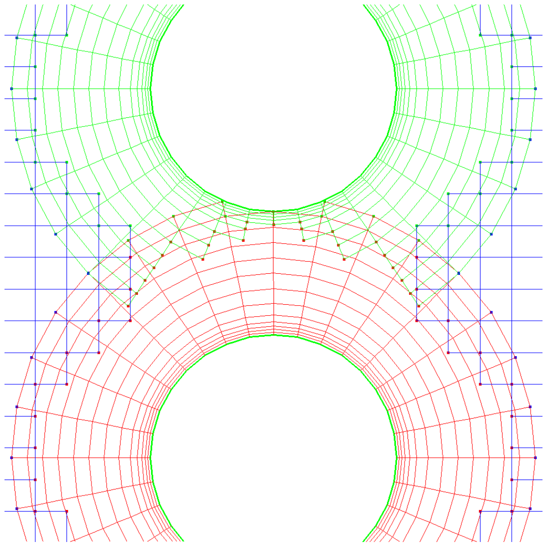


Figure 35: The lower annulus (the highest priority grid) has points that interpolate from the fine boundary layer grid of the upper annulus. This interpolation will be inaccurate if the solution varies rapidly in the boundary layer, and the lower annulus will be unable to represent the boundary layer solution accurately. This problem cannot be fixed by simply changing the priorities of the grids.

Figure (35) shows a grid where the highest priority grid (the bottom annulus) interpolates from the fine boundary layer grid of the top annulus. By turning on the flag to improve the quality of interpolation the grid shown in figure (36) results.

We use a simple measure of the quality of the interpolation to be the relative size of the grid cells on the two grids involved.

$$\text{quality of interpolation} = \frac{\text{cell size of the interpolation point}}{\text{cell size of the interpolee point}}$$

The quality is bad (i.e. large) if the interpolee grid cells are smaller. This simple measure seems adequate for our purposes of preventing coarse grid points on higher priority grids from interpolating from lower priority grids.

The **algorithm** for removing poor quality points is

1. Follow the standard algorithm until all points have been interpolated but redundant points have not yet been removed.

2. Try to interpolate all points on the finest grid that can interpolate from a lower priority grid. (This is not done in the standard case).

3. Attempt to remove poor quality points from the *boundary* of the interpolation point region where a point interpolates from a lower priority grid. A point is removed if it is not needed for discretization and the quality measure is greater than a specified value (normally around 2). If a point is removed then also check the new *boundary* points that are now exposed.

4. After points have been removed we need to go back and update any other interpolation points that can no-longer interpolate (since they required some of the points that were deleted).

The algorithm is supposed to be guaranteed to give a valid grid provided a grid could be made without the improvement steps.

9.7.1 Note:

There is a more sophisticated way to measure the quality of interpolation. ***This measure is not used currently**.

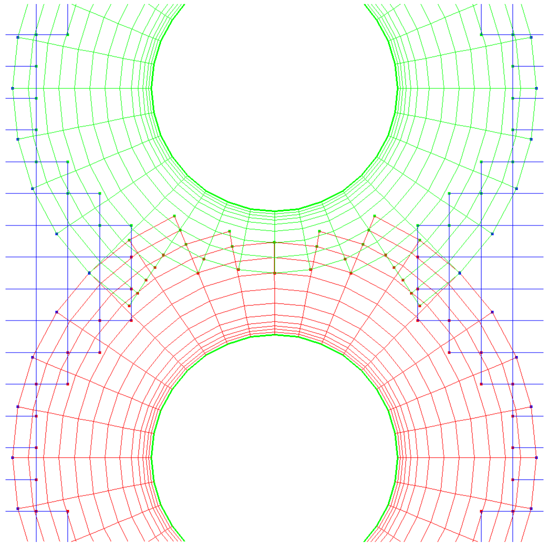


Figure 36: With the ‘improved quality’ option turned on, the lower annulus no longer interpolates from the fine boundary layer of the upper annulus.

One way to measure the quality of the interpolation is defined as follows. We would like the cell at an interpolation point on grid A to be approximately the same size, shape and orientation as the cells on the interpolee grid B. The vector

$$\mathbf{d}_i^A = \frac{\partial \mathbf{x}^A}{\partial r_i} \Delta r_i^A$$

measures the grid cell spacing and orientation of the side of the cell along the axis r_i of grid A. This vector corresponds to a vector in the parameter space of grid B given by

$$\mathbf{r}_i^B = \left[\frac{\partial \mathbf{r}^B}{\partial \mathbf{x}} \right] \mathbf{d}_i^A$$

The length in grid cells of this vector $\mathbf{r}_i^B$ is approximately

$$\left\| \begin{bmatrix} \frac{1}{\Delta r_1^B} & 0 & 0 \\ 0 & \frac{1}{\Delta r_2^B} & 0 \\ 0 & 0 & \frac{1}{\Delta r_3^B} \end{bmatrix} \mathbf{r}_i^B \right\|$$

where we have scaled each element by the appropriate grid spacing. This length should be near 1 for good quality (since the original vector $\mathbf{d}_i^A$ has a length of one grid cell).

Thus to measure the quality of all sides on the original cell we can compute

$$p = \left\| \begin{bmatrix} \frac{1}{\Delta r_1^B} & 0 & 0 \\ 0 & \frac{1}{\Delta r_2^B} & 0 \\ 0 & 0 & \frac{1}{\Delta r_3^B} \end{bmatrix} \left[\frac{\partial \mathbf{r}^B}{\partial \mathbf{x}} \right] \left[\frac{\partial \mathbf{x}^A}{\partial \mathbf{r}} \right] \begin{bmatrix} \Delta r_1^A & 0 & 0 \\ 0 & \Delta r_2^A & 0 \\ 0 & 0 & \Delta r_3^A \end{bmatrix} \right\|$$

The interpolation will be defined to be of high quality if this norm is near 1. In particular we use the quality measure

$$q = \frac{1}{2}(p + \frac{1}{p})$$

where we prefer points with a smaller value for q.

10 Treatment of nearby boundaries and the boundaryDiscretisationWidth

** new with version 18**

Figure (37) shows the grid generated in the case when two boundaries are very near to one another. The `boundaryDiscretisationWidth` parameter, which is by default 3, indicates that any boundary point that is a discretisation point should have two interior neighbouring points so that a one-sided 3-point scheme could be applied on the boundary. To ensure this condition is satisfied extra points are allowed that normally would not be valid. The interpolation points that are outside the domain are "interpolated" from the nearest point on the boundary by pretending that the interpolation point has been moved to the boundary. This will only be first order accurate interpolation.

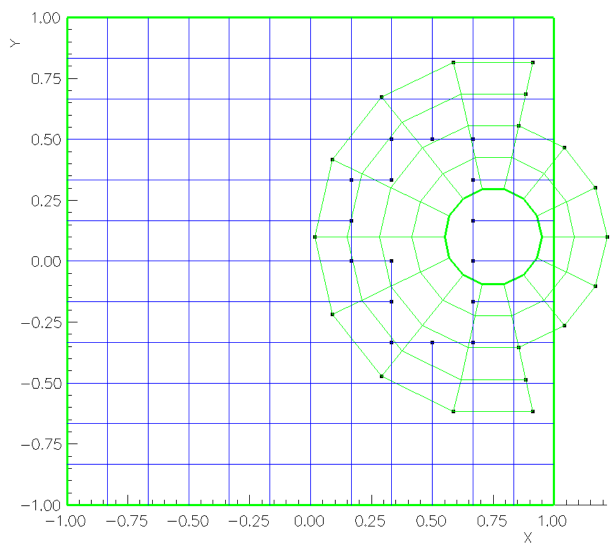


Figure 37: When two boundaries are nearby to one another the overlapping grid algorithm ensures that enough interior grid-points remain next to the boundary points to allow the boundary point to be discretised. While not very accurate this approach at least allows a grid to be built.

11 Adaptive Mesh Refinement

When refinement grids are added to an overlapping grid and a refinement grid overlaps an interpolation boundary, the Ogen function `updateRefinement` should be called. This function will cut holes in the refinement grids and determine how to interpolate points on the hole-boundary.

The order of preference for the interpolation of a point on the hole-boundary of a refinement grid is to

1. interpolate from another refinement at the same level and different base grid
2. interpolate from another refinement at a lower level and different base grid
3. interpolate from a refinement grid on the same base grid (this case should only be used as a backup and should normally not be needed).

11.1 The algorithm for updating refinement meshes added to an overlapping grid.

There are two main steps in the algorithm for adding refinement meshes to an overlapping grid.

1. Build a mask array for each refinement grid that indicates where holes are and which points should be interpolated.
2. For each interpolation point on the hole boundary, find which grid to interpolate from.

To be efficient, these steps are performed with a different procedure than the normal overlapping grid algorithm. The mask array is built entirely by looking at the mask array from the base grids. The interpolation points are determined by looking at the interpolation points on the base grids in order to determine the likely interpolee grids.

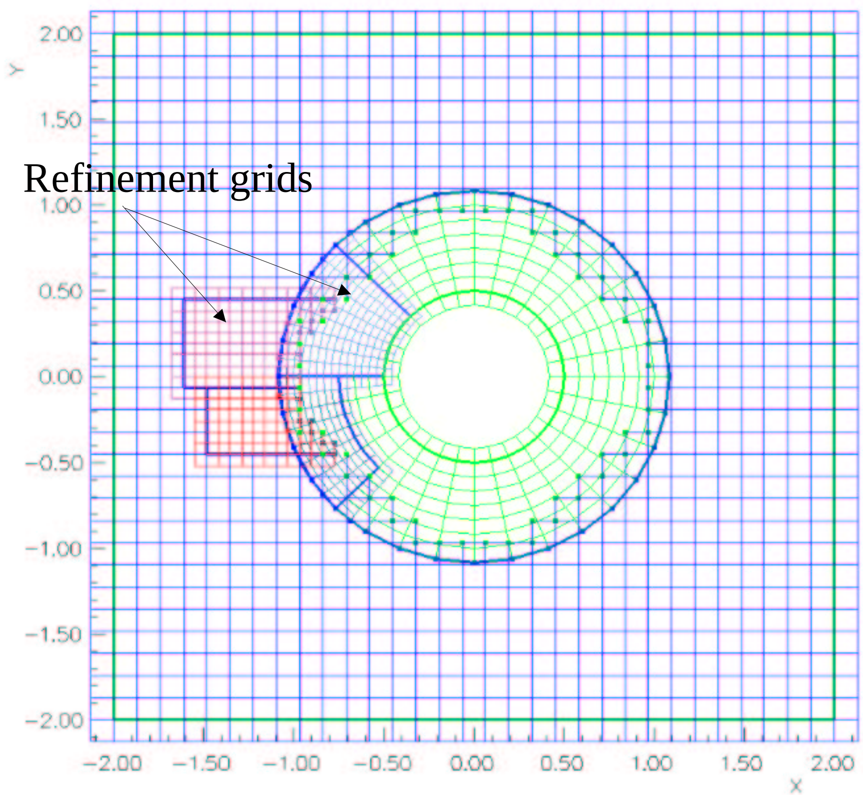
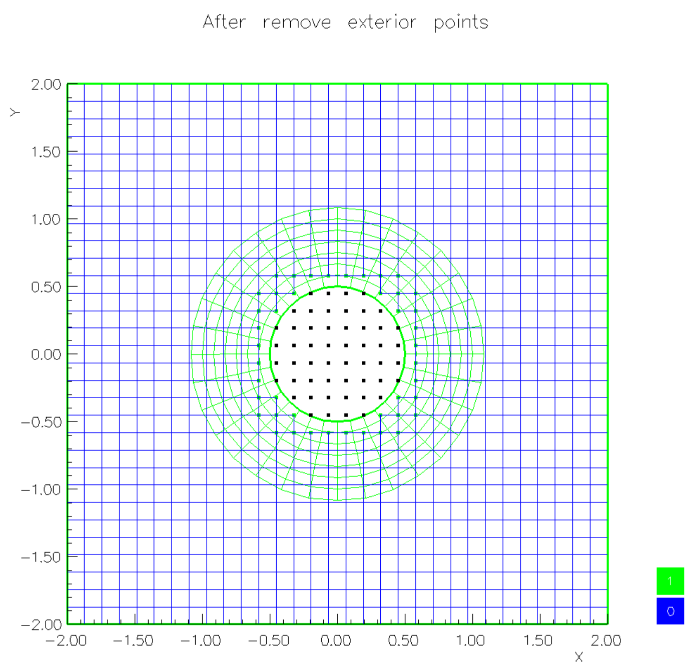


Figure 38: When refinement grids are added to an overlapping grid, the `updateRefinement` function should be called in order to compute a valid grid.

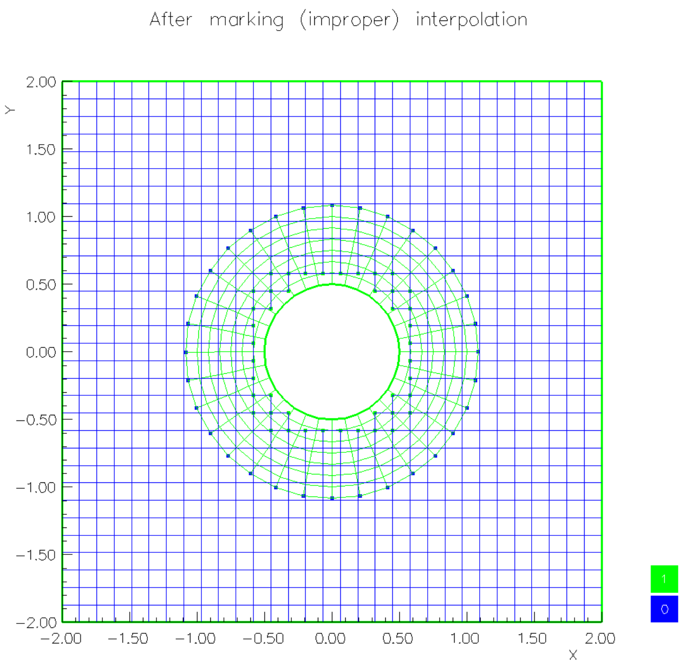
11.2 Example: Circle in a Channel

These figures show the circle in a channel grid at various stages in the overlap algorithm.

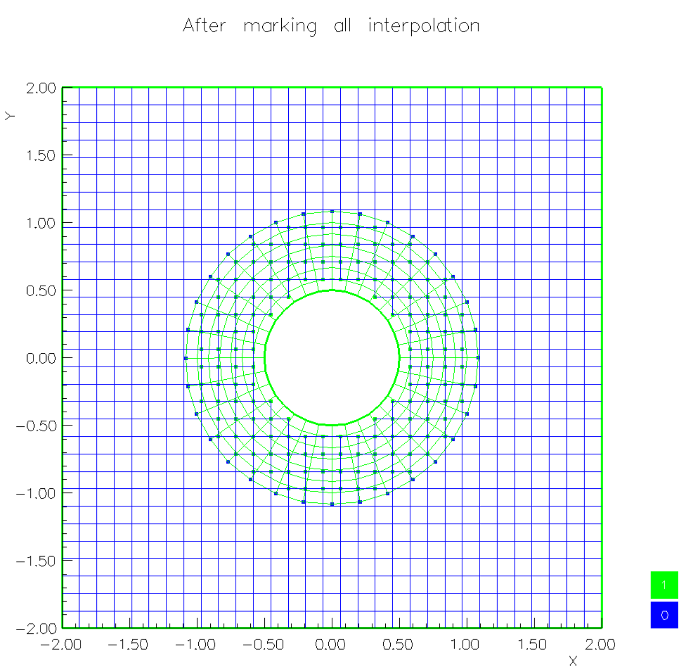
Grid after cutting holes. Physical boundaries are used to cut holes in nearby grids. The hole cutting algorithm will generate a barrier of hole points and interpolation points that bounds the entire hole region.



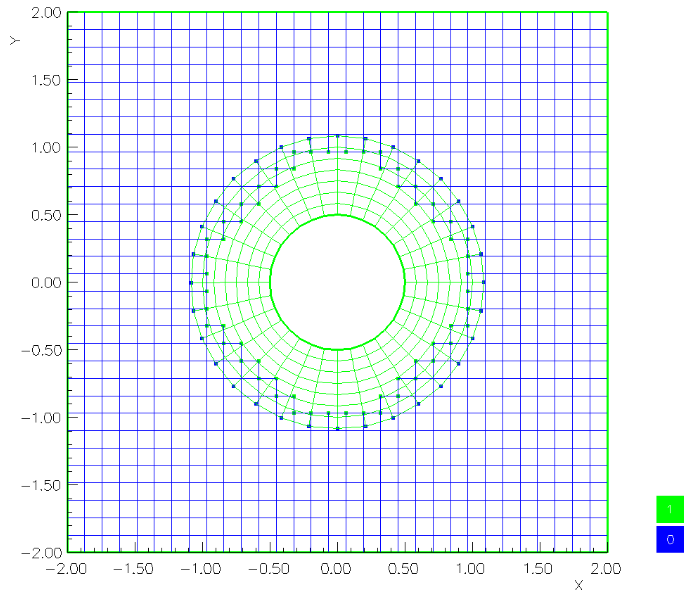
Grid after removing all exterior points. The exterior points are easily swept out after the hole boundary has been marked.



Grid after marking (improper) interpolation. These improper interpolation points need only lie inside another grid.



Grid after marking all (proper) interpolation. We have attempted to interpolate discretization points on each grid from grids of higher priority.

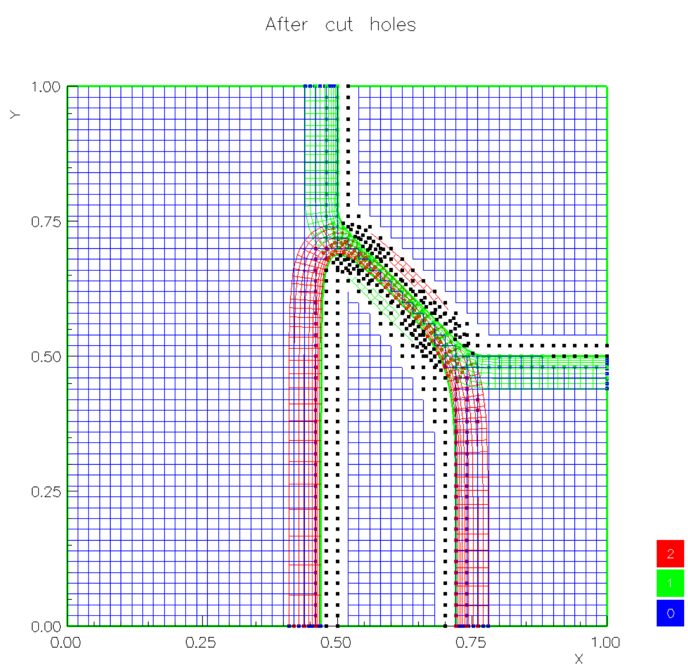


Finished grid after removing excess interpolation points.

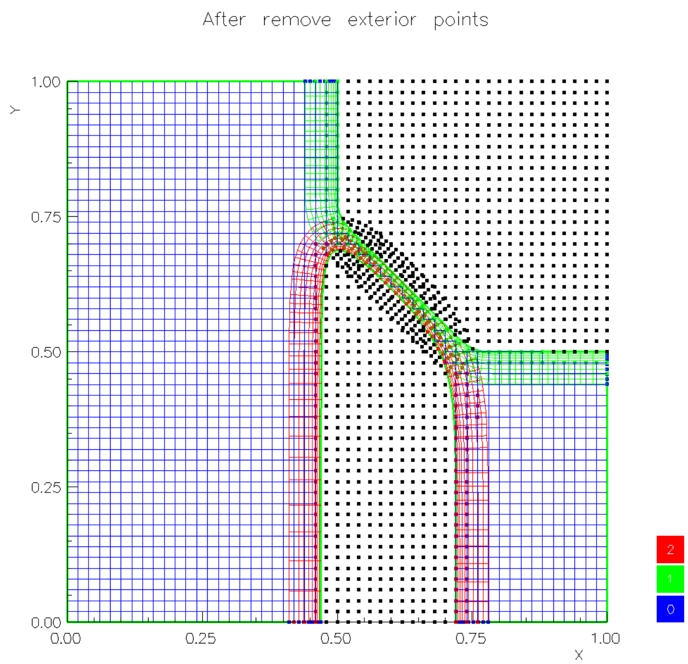
11.3 Example: Valve

These figures show the grid for a valve at various stages in the overlap algorithm.

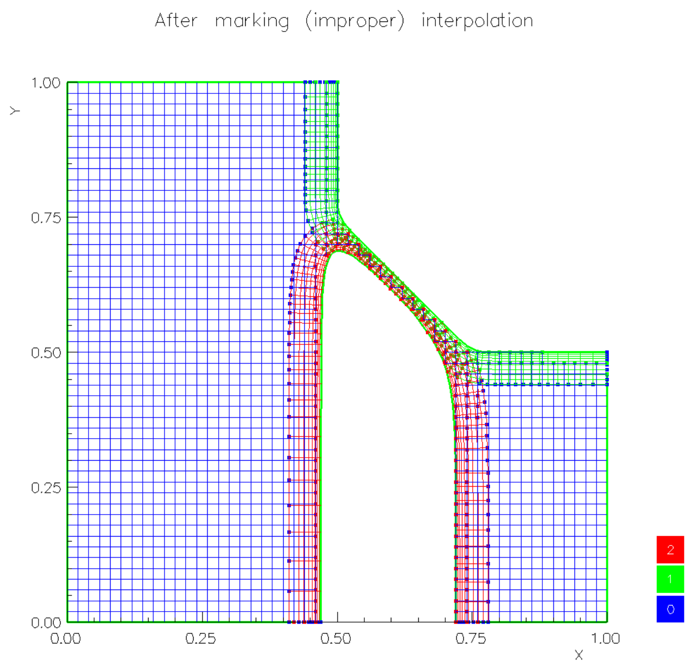
Grid after interpolation on boundaries.



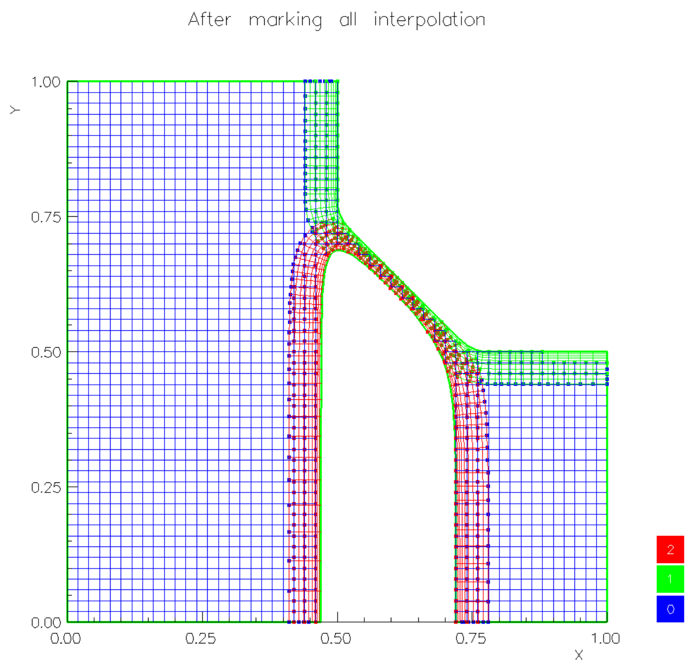
Grid after cutting holes. Physical boundaries are used to cut holes in nearby grids. The hole cutting algorithm will generate a barrier of hole points and interpolation points that bounds the entire hole region.



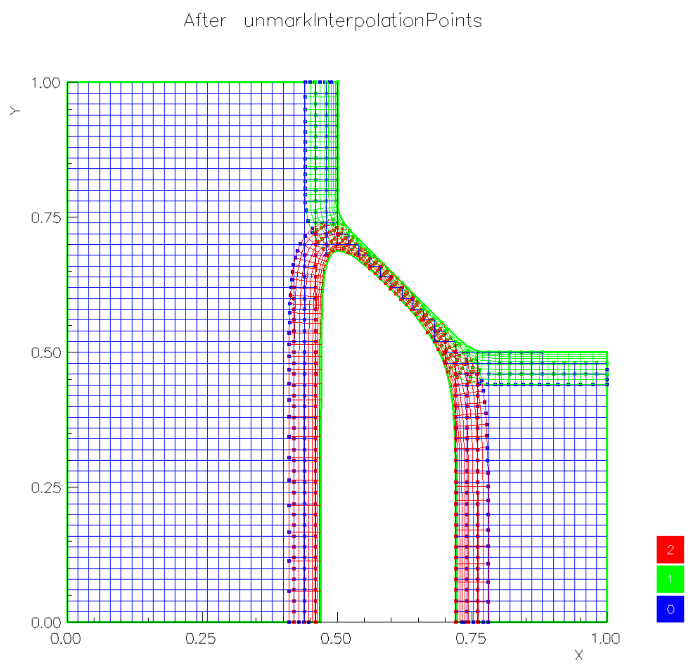
Grid after removing all exterior points. The exterior points are easily swept out after the hole boundary has been marked.



Grid after marking (improper) interpolation. These improper interpolation points need only lie inside another grid.



Grid after marking all (proper) interpolation.



Finished grid after removing excess interpolation points.

References

[1] G. CHESSHIRE AND W. HENSHAW, *Composite overlapping meshes for the solution of partial differential equations*, J. Comp. Phys., 90 (1990), pp. 1–64.

[2] W. HENSHAW, *Mappings for Overture, a description of the Mapping class and documentation for many useful Mappings*, Research Report LA-UR-96-3469, Los Alamos National Laboratory, 1996.

[3] ———, *Plotstuff: A class for plotting stuff from Overture*, Research Report LA-UR-96-3893, Los Alamos National Laboratory, 1996.

Index